Understanding the

LINUX
KERNEL

Other Linux resources from O'Reilly

Related titles

Building Embedded Linux Systems

Linux Device Drivers

Linux in a Nutshell

Linux Network Administrator's Guide

Linux Pocket Guide

Linux Security Cookbook™

Linux Server Hacks™

Linux Server Security

Running Linux

SELinux

Understanding Linux Network Internals

Linux Books Resource Center

linux.oreilly.com is a complete catalog of O'Reilly's books on Linux and Unix and related technologies, including sample chapters and code examples.

ONLamp.com is the premier site for the open source web platform: Linux, Apache, MySQL, and either Perl, Python, or PHP.

Conferences

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

O'REILLY NETWORK
Safari Bookshelf

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.

Understanding the
LINUX
KERNEL

THIRD EDITION

Daniel P. Bovet and Marco Cesati

Understanding the Linux Kernel, Third Edition
by Daniel P. Bovet and Marco Cesati

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Andy Oram
Production Editor:	Darren Kelly
Production Services:	Amy Parker
Cover Designer:	Edie Freedman
Interior Designer:	David Futato

Printing History:

November 2000:	First Edition.
December 2002:	Second Edition.
November 2005:	Third Edition.

Table of Contents

Preface

In the spring semester of 1997, we taught a course on operating systems based on Linux 2.0. The idea was to encourage students to read the source code. To achieve this, we assigned term projects consisting of making changes to the kernel and performing tests on the modified version. We also wrote course notes for our students about a few critical features of Linux such as task switching and task scheduling.

Out of this work—and with a lot of support from our O'Reilly editor Andy Oram—came the first edition of *Understanding the Linux Kernel* at the end of 2000, which covered Linux 2.2 with a few anticipations on Linux 2.4. The success encountered by this book encouraged us to continue along this line. At the end of 2002, we came out with a second edition covering Linux 2.4. You are now looking at the third edition, which covers Linux 2.6.

As in our previous experiences, we read thousands of lines of code, trying to make sense of them. After all this work, we can say that it was worth the effort. We learned a lot of things you don't find in books, and we hope we have succeeded in conveying some of this information in the following pages.

The Audience for This Book

All people curious about how Linux works and why it is so efficient will find answers here. After reading the book, you will find your way through the many thousands of lines of code, distinguishing between crucial data structures and secondary ones—in short, becoming a true Linux hacker.

Our work might be considered a guided tour of the Linux kernel: most of the significant data structures and many algorithms and programming tricks used in the kernel are discussed. In many cases, the relevant fragments of code are discussed line by line. Of course, you should have the Linux source code on hand and should be willing to expend some effort deciphering some of the functions that are not, for sake of brevity, fully described.

On another level, the book provides valuable insight to people who want to know more about the critical design issues in a modern operating system. It is not specifically addressed to system administrators or programmers; it is mostly for people who want to understand how things really work inside the machine! As with any good guide, we try to go beyond superficial features. We offer a background, such as the history of major features and the reasons why they were used.

Organization of the Material

When we began to write this book, we were faced with a critical decision: should we refer to a specific hardware platform or skip the hardware-dependent details and concentrate on the pure hardware-independent parts of the kernel?

Others books on Linux kernel internals have chosen the latter approach; we decided to adopt the former one for the following reasons:

- Efficient kernels take advantage of most available hardware features, such as addressing techniques, caches, processor exceptions, special instructions, processor control registers, and so on. If we want to convince you that the kernel indeed does quite a good job in performing a specific task, we must first tell what kind of support comes from the hardware.

- Even if a large portion of a Unix kernel source code is processor-independent and coded in C language, a small and critical part is coded in assembly language. A thorough knowledge of the kernel, therefore, requires the study of a few assembly language fragments that interact with the hardware.

When covering hardware features, our strategy is quite simple: only sketch the features that are totally hardware-driven while detailing those that need some software support. In fact, we are interested in kernel design rather than in computer architecture.

Our next step in choosing our path consisted of selecting the computer system to describe. Although Linux is now running on several kinds of personal computers and workstations, we decided to concentrate on the very popular and cheap IBM-compatible personal computers—and thus on the 80×86 microprocessors and on some support chips included in these personal computers. The term 80×86 *microprocessor* will be used in the forthcoming chapters to denote the Intel 80386, 80486, Pentium, Pentium Pro, Pentium II, Pentium III, and Pentium 4 microprocessors or compatible models. In a few cases, explicit references will be made to specific models.

One more choice we had to make was the order to follow in studying Linux components. We tried a bottom-up approach: start with topics that are hardware-dependent and end with those that are totally hardware-independent. In fact, we'll make many references to the 80×86 microprocessors in the first part of the book, while the rest of it is relatively hardware-independent. Significant exceptions are made in Chapter 13 and Chapter 14. In practice, following a bottom-up approach is not as simple as it looks, because the areas of memory management, process

management, and filesystems are intertwined; a few forward references—that is, references to topics yet to be explained—are unavoidable.

Each chapter starts with a theoretical overview of the topics covered. The material is then presented according to the bottom-up approach. We start with the data structures needed to support the functionalities described in the chapter. Then we usually move from the lowest level of functions to higher levels, often ending by showing how system calls issued by user applications are supported.

Level of Description

Linux source code for all supported architectures is contained in more than 14,000 C and assembly language files stored in about 1000 subdirectories; it consists of roughly 6 million lines of code, which occupy over 230 megabytes of disk space. Of course, this book can cover only a very small portion of that code. Just to figure out how big the Linux source is, consider that the whole source code of the book you are reading occupies less than 3 megabytes. Therefore, we would need more than 75 books like this to list all code, without even commenting on it!

So we had to make some choices about the parts to describe. This is a rough assessment of our decisions:

- We describe process and memory management fairly thoroughly.
- We cover the Virtual Filesystem and the Ext2 and Ext3 filesystems, although many functions are just mentioned without detailing the code; we do not discuss other filesystems supported by Linux.
- We describe device drivers, which account for roughly 50% of the kernel, as far as the kernel interface is concerned, but do not attempt analysis of each specific driver.

The book describes the official 2.6.11 version of the Linux kernel, which can be downloaded from the web site *http://www.kernel.org*.

Be aware that most distributions of GNU/Linux modify the official kernel to implement new features or to improve its efficiency. In a few cases, the source code provided by your favorite distribution might differ significantly from the one described in this book.

In many cases, we show fragments of the original code rewritten in an easier-to-read but less efficient way. This occurs at time-critical points at which sections of programs are often written in a mixture of hand-optimized C and assembly code. Once again, our aim is to provide some help in studying the original Linux code.

While discussing kernel code, we often end up describing the underpinnings of many familiar features that Unix programmers have heard of and about which they may be curious (shared and mapped memory, signals, pipes, symbolic links, and so on).

Overview of the Book

To make life easier, Chapter 1, *Introduction*, presents a general picture of what is inside a Unix kernel and how Linux competes against other well-known Unix systems.

The heart of any Unix kernel is memory management. Chapter 2, *Memory Addressing*, explains how 80×86 processors include special circuits to address data in memory and how Linux exploits them.

Processes are a fundamental abstraction offered by Linux and are introduced in Chapter 3, *Processes*. Here we also explain how each process runs either in an unprivileged User Mode or in a privileged Kernel Mode. Transitions between User Mode and Kernel Mode happen only through well-established hardware mechanisms called *interrupts* and *exceptions*. These are introduced in Chapter 4, *Interrupts and Exceptions*.

In many occasions, the kernel has to deal with bursts of interrupt signals coming from different devices and processors. Synchronization mechanisms are needed so that all these requests can be serviced in an interleaved way by the kernel: they are discussed in Chapter 5, *Kernel Synchronization*, for both uniprocessor and multiprocessor systems.

One type of interrupt is crucial for allowing Linux to take care of elapsed time; further details can be found in Chapter 6, *Timing Measurements*.

Chapter 7, *Process Scheduling*, explains how Linux executes, in turn, every active process in the system so that all of them can progress toward their completions.

Next we focus again on memory. Chapter 8, *Memory Management*, describes the sophisticated techniques required to handle the most precious resource in the system (besides the processors, of course): available memory. This resource must be granted both to the Linux kernel and to the user applications. Chapter 9, *Process Address Space*, shows how the kernel copes with the requests for memory issued by greedy application programs.

Chapter 10, *System Calls*, explains how a process running in User Mode makes requests to the kernel, while Chapter 11, *Signals*, describes how a process may send synchronization signals to other processes. Now we are ready to move on to another essential topic, how Linux implements the filesystem. A series of chapters cover this topic. Chapter 12, *The Virtual Filesystem*, introduces a general layer that supports many different filesystems. Some Linux files are special because they provide trapdoors to reach hardware devices; Chapter 13, *I/O Architecture and Device Drivers*, and Chapter 14, *Block Device Drivers*, offer insights on these special files and on the corresponding hardware device drivers.

Another issue to consider is disk access time; Chapter 15, *The Page Cache*, shows how a clever use of RAM reduces disk accesses, therefore improving system performance significantly. Building on the material covered in these last chapters, we can now explain in Chapter 16, *Accessing Files*, how user applications access normal files. Chapter 17, *Page Frame Reclaiming*, completes our discussion of Linux memory management and explains the techniques used by Linux to ensure that enough

memory is always available. The last chapter dealing with files is Chapter 18, *The Ext2 and Ext3 Filesystems*, which illustrates the most frequently used Linux filesystem, namely Ext2 and its recent evolution, Ext3.

The last two chapters end our detailed tour of the Linux kernel: Chapter 19, *Process Communication*, introduces communication mechanisms other than signals available to User Mode processes; Chapter 20, *Program Execution*, explains how user applications are started.

Last, but not least, are the appendixes: Appendix A, *System Startup*, sketches out how Linux is booted, while Appendix B, *Modules*, describes how to dynamically reconfigure the running kernel, adding and removing functionalities as needed. The Source Code Index includes all the Linux symbols referenced in the book; here you will find the name of the Linux file defining each symbol and the book's page number where it is explained. We think you'll find it quite handy.

Background Information

No prerequisites are required, except some skill in C programming language and perhaps some knowledge of an assembly language.

Conventions in This Book

The following is a list of typographical conventions used in this book:

Constant Width
> Used to show the contents of code files or the output from commands, and to indicate source code keywords that appear in code.

Italic
> Used for file and directory names, program and command names, command-line options, and URLs, and for emphasizing new terms.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472
> (800) 998-9938 (in the United States or Canada)
> (707) 829-0515 (international or local)
> (707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, or any additional information. You can access this page at:

> *http://www.oreilly.com/catalog/understandlk/*

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, it means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

Acknowledgments

This book would not have been written without the precious help of the many students of the University of Rome school of engineering "Tor Vergata" who took our course and tried to decipher lecture notes about the Linux kernel. Their strenuous efforts to grasp the meaning of the source code led us to improve our presentation and correct many mistakes.

Andy Oram, our wonderful editor at O'Reilly Media, deserves a lot of credit. He was the first at O'Reilly to believe in this project, and he spent a lot of time and energy deciphering our preliminary drafts. He also suggested many ways to make the book more readable, and he wrote several excellent introductory paragraphs.

We had some prestigious reviewers who read our text quite carefully. The first edition was checked by (in alphabetical order by first name) Alan Cox, Michael Kerrisk, Paul Kinzelman, Raph Levien, and Rik van Riel.

The second edition was checked by Erez Zadok, Jerry Cooperstein, John Goerzen, Michael Kerrisk, Paul Kinzelman, Rik van Riel, and Walt Smith.

This edition has been reviewed by Charles P. Wright, Clemens Buchacher, Erez Zadok, Raphael Finkel, Rik van Riel, and Robert P. J. Day. Their comments, together with those of many readers from all over the world, helped us to remove several errors and inaccuracies and have made this book stronger.

—Daniel P. Bovet
Marco Cesati
July 2005

Introduction

Linux* is a member of the large family of Unix-like operating systems. A relative new-comer experiencing sudden spectacular popularity starting in the late 1990s, Linux joins such well-known commercial Unix operating systems as System V Release 4 (SVR4), developed by AT&T (now owned by the SCO Group); the 4.4 BSD release from the University of California at Berkeley (4.4BSD); Digital UNIX from Digital Equipment Corporation (now Hewlett-Packard); AIX from IBM; HP-UX from Hewlett-Packard; Solaris from Sun Microsystems; and Mac OS X from Apple Computer, Inc. Beside Linux, a few other opensource Unix-like kernels exist, such as FreeBSD, NetBSD, and OpenBSD.

Linux was initially developed by Linus Torvalds in 1991 as an operating system for IBM-compatible personal computers based on the Intel 80386 microprocessor. Linus remains deeply involved with improving Linux, keeping it up-to-date with various hardware developments and coordinating the activity of hundreds of Linux developers around the world. Over the years, developers have worked to make Linux available on other architectures, including Hewlett-Packard's Alpha, Intel's Itanium, AMD's AMD64, PowerPC, and IBM's zSeries.

One of the more appealing benefits to Linux is that it isn't a commercial operating system: its source code under the *GNU General Public License* (*GPL*)† is open and available to anyone to study (as we will in this book); if you download the code (the official site is *http://www.kernel.org*) or check the sources on a Linux CD, you will be able to explore, from top to bottom, one of the most successful modern operating systems. This book, in fact, assumes you have the source code on hand and can apply what we say to your own explorations.

---

* LINUX® is a registered trademark of Linus Torvalds.

† The GNU project is coordinated by the Free Software Foundation, Inc. (*http://www.gnu.org*); its aim is to implement a whole operating system freely usable by everyone. The availability of a GNU C compiler has been essential for the success of the Linux project.

Technically speaking, Linux is a true Unix kernel, although it is not a full Unix operating system because it does not include all the Unix applications, such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on. However, because most of these programs are freely available under the GPL, they can be installed in every Linux-based system.

Because the Linux kernel requires so much additional software to provide a useful environment, many Linux users prefer to rely on commercial distributions, available on CD-ROM, to get the code included in a standard Unix system. Alternatively, the code may be obtained from several different sites, for instance *http://www.kernel.org*. Several distributions put the Linux source code in the */usr/src/linux* directory. In the rest of this book, all file pathnames will refer implicitly to the Linux source code directory.

Linux Versus Other Unix-Like Kernels

The various Unix-like systems on the market, some of which have a long history and show signs of archaic practices, differ in many important respects. All commercial variants were derived from either SVR4 or 4.4BSD, and all tend to agree on some common standards like IEEE's Portable Operating Systems based on Unix (*POSIX*) and X/Open's Common Applications Environment (CAE).

The current standards specify only an application programming interface (API)—that is, a well-defined environment in which user programs should run. Therefore, the standards do not impose any restriction on internal design choices of a compliant kernel.[*]

To define a common user interface, Unix-like kernels often share fundamental design ideas and features. In this respect, Linux is comparable with the other Unix-like operating systems. Reading this book and studying the Linux kernel, therefore, may help you understand the other Unix variants, too.

The 2.6 version of the Linux kernel aims to be compliant with the IEEE POSIX standard. This, of course, means that most existing Unix programs can be compiled and executed on a Linux system with very little effort or even without the need for patches to the source code. Moreover, Linux includes all the features of a modern Unix operating system, such as virtual memory, a virtual filesystem, lightweight processes, Unix signals, SVR4 interprocess communications, support for Symmetric Multiprocessor (SMP) systems, and so on.

When Linus Torvalds wrote the first kernel, he referred to some classical books on Unix internals, like Maurice Bach's *The Design of the Unix Operating System* (Prentice Hall, 1986). Actually, Linux still has some bias toward the Unix baseline

---

[*] As a matter of fact, several non-Unix operating systems, such as Windows NT and its descendents, are POSIX-compliant.

described in Bach's book (i.e., SVR2). However, Linux doesn't stick to any particular variant. Instead, it tries to adopt the best features and design choices of several different Unix kernels.

The following list describes how Linux competes against some well-known commercial Unix kernels:

Monolithic kernel
> It is a large, complex do-it-yourself program, composed of several logically different components. In this, it is quite conventional; most commercial Unix variants are monolithic. (Notable exceptions are the Apple Mac OS X and the GNU Hurd operating systems, both derived from the Carnegie-Mellon's Mach, which follow a microkernel approach.)

Compiled and statically linked traditional Unix kernels
> Most modern kernels can dynamically load and unload some portions of the kernel code (typically, device drivers), which are usually called modules. Linux's support for modules is very good, because it is able to automatically load and unload modules on demand. Among the main commercial Unix variants, only the SVR4.2 and Solaris kernels have a similar feature.

Kernel threading
> Some Unix kernels, such as Solaris and SVR4.2/MP, are organized as a set of kernel threads. A kernel thread is an execution context that can be independently scheduled; it may be associated with a user program, or it may run only some kernel functions. Context switches between kernel threads are usually much less expensive than context switches between ordinary processes, because the former usually operate on a common address space. Linux uses kernel threads in a very limited way to execute a few kernel functions periodically; however, they do not represent the basic execution context abstraction. (That's the topic of the next item.)

Multithreaded application support
> Most modern operating systems have some kind of support for multithreaded applications—that is, user programs that are designed in terms of many relatively independent execution flows that share a large portion of the application data structures. A multithreaded user application could be composed of many lightweight processes (LWP), which are processes that can operate on a common address space, common physical memory pages, common opened files, and so on. Linux defines its own version of lightweight processes, which is different from the types used on other systems such as SVR4 and Solaris. While all the commercial Unix variants of LWP are based on kernel threads, Linux regards lightweight processes as the basic execution context and handles them via the nonstandard `clone( )` system call.

Preemptive kernel

> When compiled with the "Preemptible Kernel" option, Linux 2.6 can arbitrarily interleave execution flows while they are in privileged mode. Besides Linux 2.6, a few other conventional, general-purpose Unix systems, such as Solaris and Mach 3.0, are fully preemptive kernels. SVR4.2/MP introduces some *fixed preemption points* as a method to get limited preemption capability.

Multiprocessor support

> Several Unix kernel variants take advantage of multiprocessor systems. Linux 2.6 supports symmetric multiprocessing (SMP) for different memory models, including NUMA: the system can use multiple processors and each processor can handle any task—there is no discrimination among them. Although a few parts of the kernel code are still serialized by means of a single "big kernel lock," it is fair to say that Linux 2.6 makes a near optimal use of SMP.

Filesystem

> Linux's standard filesystems come in many flavors. You can use the plain old Ext2 filesystem if you don't have specific needs. You might switch to Ext3 if you want to avoid lengthy filesystem checks after a system crash. If you'll have to deal with many small files, the ReiserFS filesystem is likely to be the best choice. Besides Ext3 and ReiserFS, several other journaling filesystems can be used in Linux; they include IBM AIX's Journaling File System (JFS) and Silicon Graphics IRIX's XFS filesystem. Thanks to a powerful object-oriented Virtual File System technology (inspired by Solaris and SVR4), porting a foreign filesystem to Linux is generally easier than porting to other kernels.

STREAMS

> Linux has no analog to the STREAMS I/O subsystem introduced in SVR4, although it is included now in most Unix kernels and has become the preferred interface for writing device drivers, terminal drivers, and network protocols.

This assessment suggests that Linux is fully competitive nowadays with commercial operating systems. Moreover, Linux has several features that make it an exciting operating system. Commercial Unix kernels often introduce new features to gain a larger slice of the market, but these features are not necessarily useful, stable, or productive. As a matter of fact, modern Unix kernels tend to be quite bloated. By contrast, Linux—together with the other open source operating systems—doesn't suffer from the restrictions and the conditioning imposed by the market, hence it can freely evolve according to the ideas of its designers (mainly Linus Torvalds). Specifically, Linux offers the following advantages over its commercial competitors:

Linux is cost-free. You can install a complete Unix system at no expense other than the hardware (of course).

Linux is fully customizable in all its components. Thanks to the compilation options, you can customize the kernel by selecting only the features really

needed. Moreover, thanks to the GPL, you are allowed to freely read and modify the source code of the kernel and of all system programs.[*]

Linux runs on low-end, inexpensive hardware platforms. You are able to build a network server using an old Intel 80386 system with 4 MB of RAM.

Linux is powerful. Linux systems are very fast, because they fully exploit the features of the hardware components. The main Linux goal is efficiency, and indeed many design choices of commercial variants, like the STREAMS I/O subsystem, have been rejected by Linus because of their implied performance penalty.

Linux developers are excellent programmers. Linux systems are very stable; they have a very low failure rate and system maintenance time.

The Linux kernel can be very small and compact. It is possible to fit a kernel image, including a few system programs, on just one 1.44 MB floppy disk. As far as we know, none of the commercial Unix variants is able to boot from a single floppy disk.

Linux is highly compatible with many common operating systems. Linux lets you directly mount filesystems for all versions of MS-DOS and Microsoft Windows, SVR4, OS/2, Mac OS X, Solaris, SunOS, NEXTSTEP, many BSD variants, and so on. Linux also is able to operate with many network layers, such as Ethernet (as well as Fast Ethernet, Gigabit Ethernet, and 10 Gigabit Ethernet), Fiber Distributed Data Interface (FDDI), High Performance Parallel Interface (HIPPI), IEEE 802.11 (Wireless LAN), and IEEE 802.15 (Bluetooth). By using suitable libraries, Linux systems are even able to directly run programs written for other operating systems. For example, Linux is able to execute some applications written for MS-DOS, Microsoft Windows, SVR3 and R4, 4.4BSD, SCO Unix, Xenix, and others on the 80x86 platform.

Linux is well supported. Believe it or not, it may be a lot easier to get patches and updates for Linux than for any proprietary operating system. The answer to a problem often comes back within a few hours after sending a message to some newsgroup or mailing list. Moreover, drivers for Linux are usually available a few weeks after new hardware products have been introduced on the market. By contrast, hardware manufacturers release device drivers for only a few commercial operating systems—usually Microsoft's. Therefore, all commercial Unix variants run on a restricted subset of hardware components.

With an estimated installed base of several tens of millions, people who are used to certain features that are standard under other operating systems are starting to expect the same from Linux. In that regard, the demand on Linux developers is also

---

[*] Many commercial companies are now supporting their products under Linux. However, many of them aren't distributed under an open source license, so you might not be allowed to read or modify their source code.

increasing. Luckily, though, Linux has evolved under the close direction of Linus and his subsystem maintainers to accommodate the needs of the masses.

Hardware Dependency

Linux tries to maintain a neat distinction between hardware-dependent and hardware-independent source code. To that end, both the *arch* and the *include* directories include 23 subdirectories that correspond to the different types of hardware platforms supported. The standard names of the platforms are:

alpha
> Hewlett-Packard's Alpha workstations (originally Digital, then Compaq; no longer manufactured)

arm, arm26
> ARM processor-based computers such as PDAs and embedded devices

cris
> "Code Reduced Instruction Set" CPUs used by Axis in its thin-servers, such as web cameras or development boards

frv
> Embedded systems based on microprocessors of the Fujitsu's FR-V family

h8300
> Hitachi h8/300 and h8S RISC 8/16-bit microprocessors

i386
> IBM-compatible personal computers based on 80**x**86 microprocessors

ia64
> Workstations based on the Intel 64-bit Itanium microprocessor

m32r
> Computers based on the Renesas M32R family of microprocessors

m68k, m68knommu
> Personal computers based on Motorola MC680×0 microprocessors

mips
> Workstations based on MIPS microprocessors, such as those marketed by Silicon Graphics

parisc
> Workstations based on Hewlett Packard HP 9000 PA-RISC microprocessors

ppc, ppc64
> Workstations based on the 32-bit and 64-bit Motorola-IBM PowerPC microprocessors

s390
> IBM ESA/390 and zSeries mainframes

sh, sh64

Embedded systems based on SuperH microprocessors developed by Hitachi and STMicroelectronics

sparc, sparc64

Workstations based on Sun Microsystems SPARC and 64-bit Ultra SPARC microprocessors

um

User Mode Linux, a virtual platform that allows developers to run a kernel in User Mode

v850

NEC V850 microcontrollers that incorporate a 32-bit RISC core based on the Harvard architecture

x86_64

Workstations based on the AMD's 64-bit microprocessors—such Athlon and Opteron—and Intel's ia32e/EM64T 64-bit microprocessors

Linux Versions

Up to kernel version 2.5, Linux identified kernels through a simple numbering scheme. Each version was characterized by three numbers, separated by periods. The first two numbers were used to identify the version; the third number identified the release. The first version number, namely 2, has stayed unchanged since 1996. The second version number identified the type of kernel: if it was even, it denoted a stable version; otherwise, it denoted a development version.

As the name suggests, stable versions were thoroughly checked by Linux distributors and kernel hackers. A new stable version was released only to address bugs and to add new device drivers. Development versions, on the other hand, differed quite significantly from one another; kernel developers were free to experiment with different solutions that occasionally lead to drastic kernel changes. Users who relied on development versions for running applications could experience unpleasant surprises when upgrading their kernel to a newer release.

During development of Linux kernel version 2.6, however, a significant change in the version numbering scheme has taken place. Basically, the second number no longer identifies stable or development versions; thus, nowadays kernel developers introduce large and significant changes in the current kernel version 2.6. A new kernel 2.7 branch will be created only when kernel developers will have to test a really disruptive change; this 2.7 branch will lead to a new current kernel version, or it will be backported to the 2.6 version, or finally it will simply be dropped as a dead end.

The new model of Linux development implies that two kernels having the same version but different release numbers—for instance, 2.6.10 and 2.6.11—can differ significantly even in core components and in fundamental algorithms. Thus, when a

new kernel release appears, it is potentially unstable and buggy. To address this problem, the kernel developers may release patched versions of any kernel, which are identified by a fourth number in the version numbering scheme. For instance, at the time this paragraph was written, the latest "stable" kernel version was 2.6.11.12.

Please be aware that the kernel version described in this book is Linux 2.6.11.

Basic Operating System Concepts

Each computer system includes a basic set of programs called the *operating system*. The most important program in the set is called the *kernel*. It is loaded into RAM when the system boots and contains many critical procedures that are needed for the system to operate. The other programs are less crucial utilities; they can provide a wide variety of interactive experiences for the user—as well as doing all the jobs the user bought the computer for—but the essential shape and capabilities of the system are determined by the kernel. The kernel provides key facilities to everything else on the system and determines many of the characteristics of higher software. Hence, we often use the term "operating system" as a synonym for "kernel."

The operating system must fulfill two main objectives:

- Interact with the hardware components, servicing all low-level programmable elements included in the hardware platform.
- Provide an execution environment to the applications that run on the computer system (the so-called user programs).

Some operating systems allow all user programs to directly play with the hardware components (a typical example is MS-DOS). In contrast, a Unix-like operating system hides all low-level details concerning the physical organization of the computer from applications run by the user. When a program wants to use a hardware resource, it must issue a request to the operating system. The kernel evaluates the request and, if it chooses to grant the resource, interacts with the proper hardware components on behalf of the user program.

To enforce this mechanism, modern operating systems rely on the availability of specific hardware features that forbid user programs to directly interact with low-level hardware components or to access arbitrary memory locations. In particular, the hardware introduces at least two different *execution modes* for the CPU: a nonprivileged mode for user programs and a privileged mode for the kernel. Unix calls these *User Mode* and *Kernel Mode*, respectively.

In the rest of this chapter, we introduce the basic concepts that have motivated the design of Unix over the past two decades, as well as Linux and other operating systems. While the concepts are probably familiar to you as a Linux user, these sections try to delve into them a bit more deeply than usual to explain the requirements they place on an operating system kernel. These broad considerations refer to virtually all

Unix-like systems. The other chapters of this book will hopefully help you understand the Linux kernel internals.

Multiuser Systems

A *multiuser system* is a computer that is able to concurrently and independently execute several applications belonging to two or more users. *Concurrently* means that applications can be active at the same time and contend for the various resources such as CPU, memory, hard disks, and so on. *Independently* means that each application can perform its task with no concern for what the applications of the other users are doing. Switching from one application to another, of course, slows down each of them and affects the response time seen by the users. Many of the complexities of modern operating system kernels, which we will examine in this book, are present to minimize the delays enforced on each program and to provide the user with responses that are as fast as possible.

Multiuser operating systems must include several features:

- An authentication mechanism for verifying the user's identity
- A protection mechanism against buggy user programs that could block other applications running in the system
- A protection mechanism against malicious user programs that could interfere with or spy on the activity of other users
- An accounting mechanism that limits the amount of resource units assigned to each user

To ensure safe protection mechanisms, operating systems must use the hardware protection associated with the CPU privileged mode. Otherwise, a user program would be able to directly access the system circuitry and overcome the imposed bounds. Unix is a multiuser system that enforces the hardware protection of system resources.

Users and Groups

In a multiuser system, each user has a private space on the machine; typically, he owns some quota of the disk space to store files, receives private mail messages, and so on. The operating system must ensure that the private portion of a user space is visible only to its owner. In particular, it must ensure that no user can exploit a system application for the purpose of violating the private space of another user.

All users are identified by a unique number called the *User ID*, or *UID*. Usually only a restricted number of persons are allowed to make use of a computer system. When one of these users starts a working session, the system asks for a *login name* and a *password*. If the user does not input a valid pair, the system denies access. Because the password is assumed to be secret, the user's privacy is ensured.

To selectively share material with other users, each user is a member of one or more *user groups*, which are identified by a unique number called a *user group ID*. Each file is associated with exactly one group. For example, access can be set so the user owning the file has read and write privileges, the group has read-only privileges, and other users on the system are denied access to the file.

Any Unix-like operating system has a special user called *root* or *superuser*. The system administrator must log in as root to handle user accounts, perform maintenance tasks such as system backups and program upgrades, and so on. The root user can do almost everything, because the operating system does not apply the usual protection mechanisms to her. In particular, the root user can access every file on the system and can manipulate every running user program.

Processes

All operating systems use one fundamental abstraction: the *process*. A process can be defined either as "an instance of a program in execution" or as the "execution context" of a running program. In traditional operating systems, a process executes a single sequence of instructions in an *address space*; the address space is the set of memory addresses that the process is allowed to reference. Modern operating systems allow processes with multiple execution flows—that is, multiple sequences of instructions executed in the same address space.

Multiuser systems must enforce an execution environment in which several processes can be active concurrently and contend for system resources, mainly the CPU. Systems that allow concurrent active processes are said to be *multiprogramming* or *multiprocessing*.[*] It is important to distinguish programs from processes; several processes can execute the same program concurrently, while the same process can execute several programs sequentially.

On uniprocessor systems, just one process can hold the CPU, and hence just one execution flow can progress at a time. In general, the number of CPUs is always restricted, and therefore only a few processes can progress at once. An operating system component called the *scheduler* chooses the process that can progress. Some operating systems allow only *nonpreemptable* processes, which means that the scheduler is invoked only when a process voluntarily relinquishes the CPU. But processes of a multiuser system must be *preemptable*; the operating system tracks how long each process holds the CPU and periodically activates the scheduler.

Unix is a multiprocessing operating system with preemptable processes. Even when no user is logged in and no application is running, several system processes monitor the peripheral devices. In particular, several processes listen at the system terminals waiting for user logins. When a user inputs a login name, the listening process runs a program that validates the user password. If the user identity is acknowledged, the

---

[*] Some multiprocessing operating systems are not multiuser; an example is Microsoft Windows 98.

process creates another process that runs a shell into which commands are entered. When a graphical display is activated, one process runs the window manager, and each window on the display is usually run by a separate process. When a user creates a graphics shell, one process runs the graphics windows and a second process runs the shell into which the user can enter the commands. For each user command, the shell process creates another process that executes the corresponding program.

Unix-like operating systems adopt a *process/kernel model*. Each process has the illusion that it's the only process on the machine, and it has exclusive access to the operating system services. Whenever a process makes a system call (i.e., a request to the kernel, see Chapter 10), the hardware changes the privilege mode from User Mode to Kernel Mode, and the process starts the execution of a kernel procedure with a strictly limited purpose. In this way, the operating system acts within the execution context of the process in order to satisfy its request. Whenever the request is fully satisfied, the kernel procedure forces the hardware to return to User Mode and the process continues its execution from the instruction following the system call.

Kernel Architecture

As stated before, most Unix kernels are monolithic: each kernel layer is integrated into the whole kernel program and runs in Kernel Mode on behalf of the current process. In contrast, *microkernel* operating systems demand a very small set of functions from the kernel, generally including a few synchronization primitives, a simple scheduler, and an interprocess communication mechanism. Several system processes that run on top of the microkernel implement other operating system–layer functions, like memory allocators, device drivers, and system call handlers.

Although academic research on operating systems is oriented toward microkernels, such operating systems are generally slower than monolithic ones, because the explicit message passing between the different layers of the operating system has a cost. However, microkernel operating systems might have some theoretical advantages over monolithic ones. Microkernels force the system programmers to adopt a modularized approach, because each operating system layer is a relatively independent program that must interact with the other layers through well-defined and clean software interfaces. Moreover, an existing microkernel operating system can be easily ported to other architectures fairly easily, because all hardware-dependent components are generally encapsulated in the microkernel code. Finally, microkernel operating systems tend to make better use of random access memory (RAM) than monolithic ones, because system processes that aren't implementing needed functionalities might be swapped out or destroyed.

To achieve many of the theoretical advantages of microkernels without introducing performance penalties, the Linux kernel offers *modules*. A module is an object file whose code can be linked to (and unlinked from) the kernel at runtime. The object code usually consists of a set of functions that implements a filesystem, a device

driver, or other features at the kernel's upper layer. The module, unlike the external layers of microkernel operating systems, does not run as a specific process. Instead, it is executed in Kernel Mode on behalf of the current process, like any other statically linked kernel function.

The main advantages of using modules include:

A modularized approach
> Because any module can be linked and unlinked at runtime, system programmers must introduce well-defined software interfaces to access the data structures handled by modules. This makes it easy to develop new modules.

Platform independence
> Even if it may rely on some specific hardware features, a module doesn't depend on a fixed hardware platform. For example, a disk driver module that relies on the SCSI standard works as well on an IBM-compatible PC as it does on Hewlett-Packard's Alpha.

Frugal main memory usage
> A module can be linked to the running kernel when its functionality is required and unlinked when it is no longer useful; this is quite useful for small embedded systems.

No performance penalty
> Once linked in, the object code of a module is equivalent to the object code of the statically linked kernel. Therefore, no explicit message passing is required when the functions of the module are invoked.[*]

An Overview of the Unix Filesystem

The Unix operating system design is centered on its filesystem, which has several interesting characteristics. We'll review the most significant ones, since they will be mentioned quite often in forthcoming chapters.

Files

A Unix file is an information container structured as a sequence of bytes; the kernel does not interpret the contents of a file. Many programming libraries implement higher-level abstractions, such as records structured into fields and record addressing based on keys. However, the programs in these libraries must rely on system calls offered by the kernel. From the user's point of view, files are organized in a tree-structured namespace, as shown in Figure 1-1.

---

[*] A small performance penalty occurs when the module is linked and unlinked. However, this penalty can be compared to the penalty caused by the creation and deletion of system processes in microkernel operating systems.
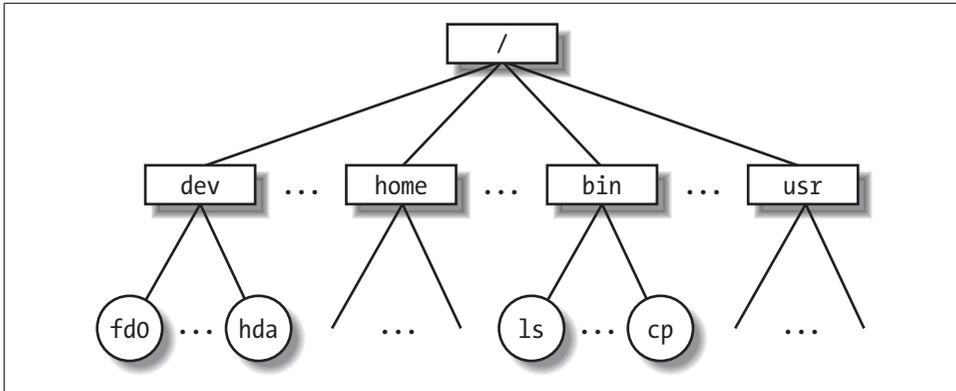
Figure 1-1. An example of a directory tree

All the nodes of the tree, except the leaves, denote directory names. A directory node contains information about the files and directories just beneath it. A file or directory name consists of a sequence of arbitrary ASCII characters,* with the exception of / and of the null character \0. Most filesystems place a limit on the length of a filename, typically no more than 255 characters. The directory corresponding to the root of the tree is called the *root directory*. By convention, its name is a slash (/). Names must be different within the same directory, but the same name may be used in different directories.

Unix associates a *current working directory* with each process (see the section "The Process/Kernel Model" later in this chapter); it belongs to the process execution context, and it identifies the directory currently used by the process. To identify a specific file, the process uses a *pathname*, which consists of slashes alternating with a sequence of directory names that lead to the file. If the first item in the pathname is a slash, the pathname is said to be *absolute*, because its starting point is the root directory. Otherwise, if the first item is a directory name or filename, the pathname is said to be *relative*, because its starting point is the process's current directory.

While specifying filenames, the notations "." and ".." are also used. They denote the current working directory and its parent directory, respectively. If the current working directory is the root directory, "." and ".." coincide.

---

* Some operating systems allow filenames to be expressed in many different alphabets, based on 16-bit extended coding of graphical characters such as Unicode.

Hard and Soft Links

A filename included in a directory is called a file *hard link*, or more simply, a *link*. The same file may have several links included in the same directory or in different ones, so it may have several filenames.

The Unix command:

```
$ ln p1 p2
```

is used to create a new hard link that has the pathname p2 for a file identified by the pathname p1.

Hard links have two limitations:

- It is not possible to create hard links for directories. Doing so might transform the directory tree into a graph with cycles, thus making it impossible to locate a file according to its name.
- Links can be created only among files included in the same filesystem. This is a serious limitation, because modern Unix systems may include several filesystems located on different disks and/or partitions, and users may be unaware of the physical divisions between them.

To overcome these limitations, *soft links* (also called *symbolic links*) were introduced a long time ago. Symbolic links are short files that contain an arbitrary pathname of another file. The pathname may refer to any file or directory located in any filesystem; it may even refer to a nonexistent file.

The Unix command:

```
$ ln -s p1 p2
```

creates a new soft link with pathname p2 that refers to pathname p1. When this command is executed, the filesystem extracts the directory part of p2 and creates a new entry in that directory of type symbolic link, with the name indicated by p2. This new file contains the name indicated by pathname p1. This way, each reference to p2 can be translated automatically into a reference to p1.

File Types

Unix files may have one of the following types:

- Regular file
- Directory
- Symbolic link
- Block-oriented device file
- Character-oriented device file
- Pipe and named pipe (also called FIFO)
- Socket

The first three file types are constituents of any Unix filesystem. Their implementation is described in detail in Chapter 18.

Device files are related both to I/O devices, and to device drivers integrated into the kernel. For example, when a program accesses a device file, it acts directly on the I/O device associated with that file (see Chapter 13).

Pipes and sockets are special files used for interprocess communication (see the section "Synchronization and Critical Regions" later in this chapter; also see Chapter 19).

File Descriptor and Inode

Unix makes a clear distinction between the contents of a file and the information about a file. With the exception of device files and files of special filesystems, each file consists of a sequence of bytes. The file does not include any control information, such as its length or an end-of-file (EOF) delimiter.

All information needed by the filesystem to handle a file is included in a data structure called an *inode*. Each file has its own inode, which the filesystem uses to identify the file.

While filesystems and the kernel functions handling them can vary widely from one Unix system to another, they must always provide at least the following attributes, which are specified in the POSIX standard:

- File type (see the previous section)
- Number of hard links associated with the file
- File length in bytes
- Device ID (i.e., an identifier of the device containing the file)
- Inode number that identifies the file within the filesystem
- UID of the file owner
- User group ID of the file
- Several timestamps that specify the inode status change time, the last access time, and the last modify time
- Access rights and file mode (see the next section)

Access Rights and File Mode

The potential users of a file fall into three classes:

- The user who is the owner of the file
- The users who belong to the same group as the file, not including the owner
- All remaining users (others)

There are three types of access rights—*read*, *write*, and *execute*—for each of these three classes. Thus, the set of access rights associated with a file consists of nine different binary flags. Three additional flags, called *suid* (*Set User ID*), *sgid* (*Set Group ID*), and *sticky*, define the file mode. These flags have the following meanings when applied to executable files:

suid

A process executing a file normally keeps the User ID (UID) of the process owner. However, if the executable file has the suid flag set, the process gets the UID of the file owner.

sgid

A process executing a file keeps the user group ID of the process group. However, if the executable file has the sgid flag set, the process gets the user group ID of the file.

sticky

An executable file with the sticky flag set corresponds to a request to the kernel to keep the program in memory after its execution terminates.[*]

When a file is created by a process, its owner ID is the UID of the process. Its owner user group ID can be either the process group ID of the creator process or the user group ID of the parent directory, depending on the value of the sgid flag of the parent directory.

File-Handling System Calls

When a user accesses the contents of either a regular file or a directory, he actually accesses some data stored in a hardware block device. In this sense, a filesystem is a user-level view of the physical organization of a hard disk partition. Because a process in User Mode cannot directly interact with the low-level hardware components, each actual file operation must be performed in Kernel Mode. Therefore, the Unix operating system defines several system calls related to file handling.

All Unix kernels devote great attention to the efficient handling of hardware block devices to achieve good overall system performance. In the chapters that follow, we will describe topics related to file handling in Linux and specifically how the kernel reacts to file-related system calls. To understand those descriptions, you will need to know how the main file-handling system calls are used; these are described in the next section.

Opening a file

Processes can access only "opened" files. To open a file, the process invokes the system call:

```
fd = open(path, flag, mode)
```

---

[*] This flag has become obsolete; other approaches based on sharing of code pages are now used (see Chapter 9).

The three parameters have the following meanings:

path
> Denotes the pathname (relative or absolute) of the file to be opened.

flag
> Specifies how the file must be opened (e.g., read, write, read/write, append). It also can specify whether a nonexisting file should be created.

mode
> Specifies the access rights of a newly created file.

This system call creates an "open file" object and returns an identifier called a *file descriptor*. An open file object contains:

- Some file-handling data structures, such as a set of flags specifying how the file has been opened, an offset field that denotes the current position in the file from which the next operation will take place (the so-called *file pointer*), and so on.
- Some pointers to kernel functions that the process can invoke. The set of permitted functions depends on the value of the flag parameter.

We discuss open file objects in detail in Chapter 12. Let's limit ourselves here to describing some general properties specified by the POSIX semantics.

- A file descriptor represents an interaction between a process and an opened file, while an open file object contains data related to that interaction. The same open file object may be identified by several file descriptors in the same process.
- Several processes may concurrently open the same file. In this case, the filesystem assigns a separate file descriptor to each file, along with a separate open file object. When this occurs, the Unix filesystem does not provide any kind of synchronization among the I/O operations issued by the processes on the same file. However, several system calls such as flock() are available to allow processes to synchronize themselves on the entire file or on portions of it (see Chapter 12).

To create a new file, the process also may invoke the creat() system call, which is handled by the kernel exactly like open().

Accessing an opened file

Regular Unix files can be addressed either sequentially or randomly, while device files and named pipes are usually accessed sequentially. In both kinds of access, the kernel stores the file pointer in the open file object—that is, the current position at which the next read or write operation will take place.

Sequential access is implicitly assumed: the read() and write() system calls always refer to the position of the current file pointer. To modify the value, a program must explicitly invoke the lseek() system call. When a file is opened, the kernel sets the file pointer to the position of the first byte in the file (offset 0).

The lseek() system call requires the following parameters:

```
newoffset = lseek(fd, offset, whence);
```

which have the following meanings:

fd
> Indicates the file descriptor of the opened file

offset
> Specifies a signed integer value that will be used for computing the new position of the file pointer

whence
> Specifies whether the new position should be computed by adding the offset value to the number 0 (offset from the beginning of the file), the current file pointer, or the position of the last byte (offset from the end of the file)

The read() system call requires the following parameters:

```
nread = read(fd, buf, count);
```

which have the following meanings:

fd
> Indicates the file descriptor of the opened file

buf
> Specifies the address of the buffer in the process's address space to which the data will be transferred

count
> Denotes the number of bytes to read

When handling such a system call, the kernel attempts to read count bytes from the file having the file descriptor fd, starting from the current value of the opened file's offset field. In some cases—end-of-file, empty pipe, and so on—the kernel does not succeed in reading all count bytes. The returned nread value specifies the number of bytes effectively read. The file pointer also is updated by adding nread to its previous value. The write() parameters are similar.

Closing a file

When a process does not need to access the contents of a file anymore, it can invoke the system call:

```
res = close(fd);
```

which releases the open file object corresponding to the file descriptor fd. When a process terminates, the kernel closes all its remaining opened files.

Renaming and deleting a file

To rename or delete a file, a process does not need to open it. Indeed, such operations do not act on the contents of the affected file, but rather on the contents of one or more directories. For example, the system call:

```
res = rename(oldpath, newpath);
```

changes the name of a file link, while the system call:

```
res = unlink(pathname);
```

decreases the file link count and removes the corresponding directory entry. The file is deleted only when the link count assumes the value 0.

An Overview of Unix Kernels

Unix kernels provide an execution environment in which applications may run. Therefore, the kernel must implement a set of services and corresponding interfaces. Applications use those interfaces and do not usually interact directly with hardware resources.

The Process/Kernel Model

As already mentioned, a CPU can run in either User Mode or Kernel Mode. Actually, some CPUs can have more than two execution states. For instance, the 80×86 microprocessors have four different execution states. But all standard Unix kernels use only Kernel Mode and User Mode.

When a program is executed in User Mode, it cannot directly access the kernel data structures or the kernel programs. When an application executes in Kernel Mode, however, these restrictions no longer apply. Each CPU model provides special instructions to switch from User Mode to Kernel Mode and vice versa. A program usually executes in User Mode and switches to Kernel Mode only when requesting a service provided by the kernel. When the kernel has satisfied the program's request, it puts the program back in User Mode.

Processes are dynamic entities that usually have a limited life span within the system. The task of creating, eliminating, and synchronizing the existing processes is delegated to a group of routines in the kernel.

The kernel itself is not a process but a process manager. The process/kernel model assumes that processes that require a kernel service use specific programming constructs called *system calls*. Each system call sets up the group of parameters that identifies the process request and then executes the hardware-dependent CPU instruction to switch from User Mode to Kernel Mode.

Besides user processes, Unix systems include a few privileged processes called *kernel threads* with the following characteristics:

- They run in Kernel Mode in the kernel address space.
- They do not interact with users, and thus do not require terminal devices.
- They are usually created during system startup and remain alive until the system is shut down.

On a uniprocessor system, only one process is running at a time, and it may run either in User or in Kernel Mode. If it runs in Kernel Mode, the processor is executing some kernel routine. Figure 1-2 illustrates examples of transitions between User and Kernel Mode. Process 1 in User Mode issues a system call, after which the process switches to Kernel Mode, and the system call is serviced. Process 1 then resumes execution in User Mode until a timer interrupt occurs, and the scheduler is activated in Kernel Mode. A process switch takes place, and Process 2 starts its execution in User Mode until a hardware device raises an interrupt. As a consequence of the interrupt, Process 2 switches to Kernel Mode and services the interrupt.
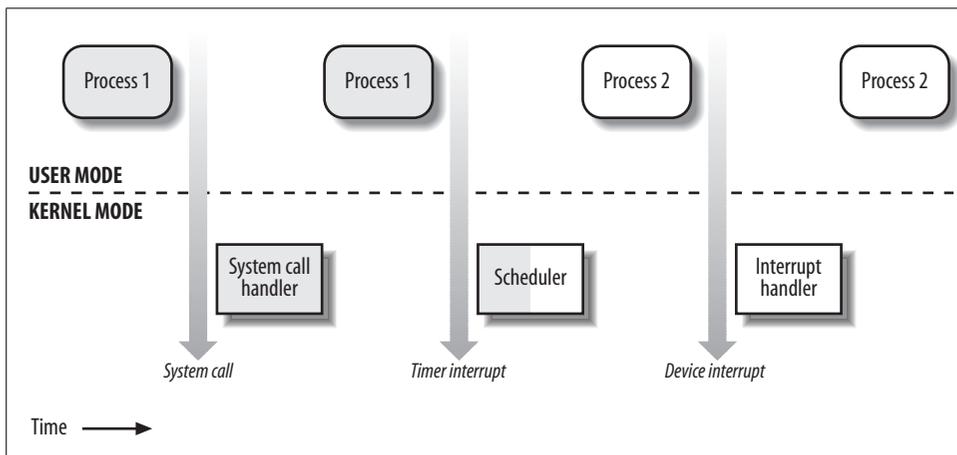

Figure 1-2. Transitions between User and Kernel Mode

Unix kernels do much more than handle system calls; in fact, kernel routines can be activated in several ways:

- A process invokes a system call.
- The CPU executing the process signals an *exception*, which is an unusual condition such as an invalid instruction. The kernel handles the exception on behalf of the process that caused it.
- A peripheral device issues an *interrupt* signal to the CPU to notify it of an event such as a request for attention, a status change, or the completion of an I/O operation. Each interrupt signal is dealt by a kernel program called an *interrupt*

handler. Because peripheral devices operate asynchronously with respect to the CPU, interrupts occur at unpredictable times.

- A kernel thread is executed. Because it runs in Kernel Mode, the corresponding program must be considered part of the kernel.

Process Implementation

To let the kernel manage processes, each process is represented by a *process descriptor* that includes information about the current state of the process.

When the kernel stops the execution of a process, it saves the current contents of several processor registers in the process descriptor. These include:

- The program counter (PC) and stack pointer (SP) registers
- The general purpose registers
- The floating point registers
- The processor control registers (Processor Status Word) containing information about the CPU state
- The memory management registers used to keep track of the RAM accessed by the process

When the kernel decides to resume executing a process, it uses the proper process descriptor fields to load the CPU registers. Because the stored value of the program counter points to the instruction following the last instruction executed, the process resumes execution at the point where it was stopped.

When a process is not executing on the CPU, it is waiting for some event. Unix kernels distinguish many wait states, which are usually implemented by queues of process descriptors; each (possibly empty) queue corresponds to the set of processes waiting for a specific event.

Reentrant Kernels

All Unix kernels are *reentrant*. This means that several processes may be executing in Kernel Mode at the same time. Of course, on uniprocessor systems, only one process can progress, but many can be blocked in Kernel Mode when waiting for the CPU or the completion of some I/O operation. For instance, after issuing a read to a disk on behalf of a process, the kernel lets the disk controller handle it and resumes executing other processes. An interrupt notifies the kernel when the device has satisfied the read, so the former process can resume the execution.

One way to provide reentrancy is to write functions so that they modify only local variables and do not alter global data structures. Such functions are called *reentrant functions*. But a reentrant kernel is not limited only to such reentrant functions (although that is how some real-time kernels are implemented). Instead, the kernel

can include nonreentrant functions and use locking mechanisms to ensure that only one process can execute a nonreentrant function at a time.

If a hardware interrupt occurs, a reentrant kernel is able to suspend the current running process even if that process is in Kernel Mode. This capability is very important, because it improves the throughput of the device controllers that issue interrupts. Once a device has issued an interrupt, it waits until the CPU acknowledges it. If the kernel is able to answer quickly, the device controller will be able to perform other tasks while the CPU handles the interrupt.

Now let's look at kernel reentrancy and its impact on the organization of the kernel. A *kernel control path* denotes the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt.

In the simplest case, the CPU executes a kernel control path sequentially from the first instruction to the last. When one of the following events occurs, however, the CPU interleaves the kernel control paths:

- A process executing in User Mode invokes a system call, and the corresponding kernel control path verifies that the request cannot be satisfied immediately; it then invokes the scheduler to select a new process to run. As a result, a process switch occurs. The first kernel control path is left unfinished, and the CPU resumes the execution of some other kernel control path. In this case, the two control paths are executed on behalf of two different processes.

- The CPU detects an exception—for example, access to a page not present in RAM—while running a kernel control path. The first control path is suspended, and the CPU starts the execution of a suitable procedure. In our example, this type of procedure can allocate a new page for the process and read its contents from disk. When the procedure terminates, the first control path can be resumed. In this case, the two control paths are executed on behalf of the same process.

- A hardware interrupt occurs while the CPU is running a kernel control path with the interrupts enabled. The first kernel control path is left unfinished, and the CPU starts processing another kernel control path to handle the interrupt. The first kernel control path resumes when the interrupt handler terminates. In this case, the two kernel control paths run in the execution context of the same process, and the total system CPU time is accounted to it. However, the interrupt handler doesn't necessarily operate on behalf of the process.

- An interrupt occurs while the CPU is running with kernel preemption enabled, and a higher priority process is runnable. In this case, the first kernel control path is left unfinished, and the CPU resumes executing another kernel control path on behalf of the higher priority process. This occurs only if the kernel has been compiled with kernel preemption support.

Figure 1-3 illustrates a few examples of noninterleaved and interleaved kernel control paths. Three different CPU states are considered:

- Running a process in User Mode (User)
- Running an exception or a system call handler (Excp)
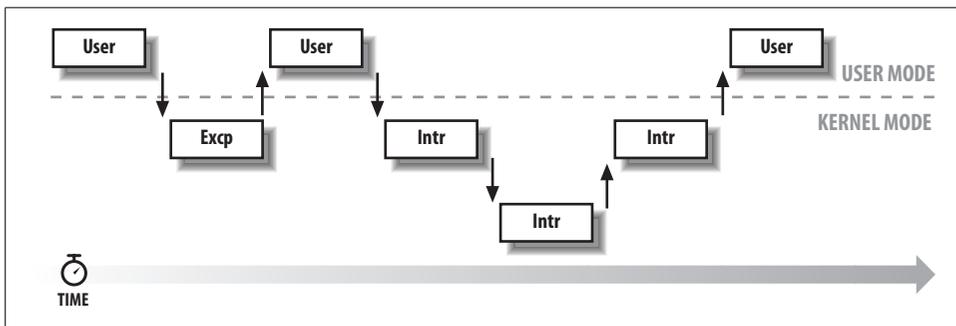- Running an interrupt handler (Intr)

Figure 1-3. Interleaving of kernel control paths

Process Address Space

Each process runs in its private address space. A process running in User Mode refers to private stack, data, and code areas. When running in Kernel Mode, the process addresses the kernel data and code areas and uses another private stack.

Because the kernel is reentrant, several kernel control paths—each related to a different process—may be executed in turn. In this case, each kernel control path refers to its own private kernel stack.

While it appears to each process that it has access to a private address space, there are times when part of the address space is shared among processes. In some cases, this sharing is explicitly requested by processes; in others, it is done automatically by the kernel to reduce memory usage.

If the same program, say an editor, is needed simultaneously by several users, the program is loaded into memory only once, and its instructions can be shared by all of the users who need it. Its data, of course, must not be shared, because each user will have separate data. This kind of shared address space is done automatically by the kernel to save memory.

Processes also can share parts of their address space as a kind of interprocess communication, using the "shared memory" technique introduced in System V and supported by Linux.

Finally, Linux supports the mmap( ) system call, which allows part of a file or the information stored on a block device to be mapped into a part of a process address space. Memory mapping can provide an alternative to normal reads and writes for transferring data. If the same file is shared by several processes, its memory mapping is included in the address space of each of the processes that share it.

Synchronization and Critical Regions

Implementing a reentrant kernel requires the use of synchronization. If a kernel control path is suspended while acting on a kernel data structure, no other kernel control path should be allowed to act on the same data structure unless it has been reset to a consistent state. Otherwise, the interaction of the two control paths could corrupt the stored information.

For example, suppose a global variable V contains the number of available items of some system resource. The first kernel control path, A, reads the variable and determines that there is just one available item. At this point, another kernel control path, B, is activated and reads the same variable, which still contains the value 1. Thus, B decreases V and starts using the resource item. Then A resumes the execution; because it has already read the value of V, it assumes that it can decrease V and take the resource item, which B already uses. As a final result, V contains –1, and two kernel control paths use the same resource item with potentially disastrous effects.

When the outcome of a computation depends on how two or more processes are scheduled, the code is incorrect. We say that there is a *race condition*.

In general, safe access to a global variable is ensured by using *atomic operations*. In the previous example, data corruption is not possible if the two control paths read and decrease V with a single, noninterruptible operation. However, kernels contain many data structures that cannot be accessed with a single operation. For example, it usually isn't possible to remove an element from a linked list with a single operation, because the kernel needs to access at least two pointers at once. Any section of code that should be finished by each process that begins it before another process can enter it is called a *critical region*.*

These problems occur not only among kernel control paths but also among processes sharing common data. Several synchronization techniques have been adopted. The following section concentrates on how to synchronize kernel control paths.

Kernel preemption disabling

To provide a drastically simple solution to synchronization problems, some traditional Unix kernels are nonpreemptive: when a process executes in Kernel Mode, it cannot be arbitrarily suspended and substituted with another process. Therefore, on a uniprocessor system, all kernel data structures that are not updated by interrupts or exception handlers are safe for the kernel to access.

Of course, a process in Kernel Mode can voluntarily relinquish the CPU, but in this case, it must ensure that all data structures are left in a consistent state. Moreover, when it resumes its execution, it must recheck the value of any previously accessed data structures that could be changed.

---

* Synchronization problems have been fully described in other works; we refer the interested reader to books on the Unix operating systems (see the Bibliography).

A synchronization mechanism applicable to preemptive kernels consists of disabling kernel preemption before entering a critical region and reenabling it right after leaving the region.

Nonpreemptability is not enough for multiprocessor systems, because two kernel control paths running on different CPUs can concurrently access the same data structure.

Interrupt disabling

Another synchronization mechanism for uniprocessor systems consists of disabling all hardware interrupts before entering a critical region and reenabling them right after leaving it. This mechanism, while simple, is far from optimal. If the critical region is large, interrupts can remain disabled for a relatively long time, potentially causing all hardware activities to freeze.

Moreover, on a multiprocessor system, disabling interrupts on the local CPU is not sufficient, and other synchronization techniques must be used.

Semaphores

A widely used mechanism, effective in both uniprocessor and multiprocessor systems, relies on the use of *semaphores*. A semaphore is simply a counter associated with a data structure; it is checked by all kernel threads before they try to access the data structure. Each semaphore may be viewed as an object composed of:

- An integer variable
- A list of waiting processes
- Two atomic methods: down( ) and up( )

The down( ) method decreases the value of the semaphore. If the new value is less than 0, the method adds the running process to the semaphore list and then blocks (i.e., invokes the scheduler). The up( ) method increases the value of the semaphore and, if its new value is greater than or equal to 0, reactivates one or more processes in the semaphore list.

Each data structure to be protected has its own semaphore, which is initialized to 1. When a kernel control path wishes to access the data structure, it executes the down( ) method on the proper semaphore. If the value of the new semaphore isn't negative, access to the data structure is granted. Otherwise, the process that is executing the kernel control path is added to the semaphore list and blocked. When another process executes the up( ) method on that semaphore, one of the processes in the semaphore list is allowed to proceed.

Spin locks

In multiprocessor systems, semaphores are not always the best solution to the synchronization problems. Some kernel data structures should be protected from being

concurrently accessed by kernel control paths that run on different CPUs. In this case, if the time required to update the data structure is short, a semaphore could be very inefficient. To check a semaphore, the kernel must insert a process in the semaphore list and then suspend it. Because both operations are relatively expensive, in the time it takes to complete them, the other kernel control path could have already released the semaphore.

In these cases, multiprocessor operating systems use *spin locks*. A spin lock is very similar to a semaphore, but it has no process list; when a process finds the lock closed by another process, it "spins" around repeatedly, executing a tight instruction loop until the lock becomes open.

Of course, spin locks are useless in a uniprocessor environment. When a kernel control path tries to access a locked data structure, it starts an endless loop. Therefore, the kernel control path that is updating the protected data structure would not have a chance to continue the execution and release the spin lock. The final result would be that the system hangs.

Avoiding deadlocks

Processes or kernel control paths that synchronize with other control paths may easily enter a *deadlock* state. The simplest case of deadlock occurs when process *p1* gains access to data structure *a* and process *p2* gains access to *b*, but *p1* then waits for *b* and *p2* waits for *a*. Other more complex cyclic waits among groups of processes also may occur. Of course, a deadlock condition causes a complete freeze of the affected processes or kernel control paths.

As far as kernel design is concerned, deadlocks become an issue when the number of kernel locks used is high. In this case, it may be quite difficult to ensure that no deadlock state will ever be reached for all possible ways to interleave kernel control paths. Several operating systems, including Linux, avoid this problem by requesting locks in a predefined order.

Signals and Interprocess Communication

Unix *signals* provide a mechanism for notifying processes of system events. Each event has its own signal number, which is usually referred to by a symbolic constant such as `SIGTERM`. There are two kinds of system events:

Asynchronous notifications
> For instance, a user can send the interrupt signal `SIGINT` to a foreground process by pressing the interrupt keycode (usually Ctrl-C) at the terminal.

Synchronous notifications
> For instance, the kernel sends the signal `SIGSEGV` to a process when it accesses a memory location at an invalid address.

The POSIX standard defines about 20 different signals, 2 of which are user-definable and may be used as a primitive mechanism for communication and synchronization among processes in User Mode. In general, a process may react to a signal delivery in two possible ways:

- Ignore the signal.
- Asynchronously execute a specified procedure (the signal handler).

If the process does not specify one of these alternatives, the kernel performs a *default action* that depends on the signal number. The five possible default actions are:

- Terminate the process.
- Write the execution context and the contents of the address space in a file (*core dump*) and terminate the process.
- Ignore the signal.
- Suspend the process.
- Resume the process's execution, if it was stopped.

Kernel signal handling is rather elaborate, because the POSIX semantics allows processes to temporarily block signals. Moreover, the SIGKILL and SIGSTOP signals cannot be directly handled by the process or ignored.

AT&T's Unix System V introduced other kinds of interprocess communication among processes in User Mode, which have been adopted by many Unix kernels: *semaphores*, *message queues*, and *shared memory*. They are collectively known as *System V IPC*.

The kernel implements these constructs as *IPC resources*. A process acquires a resource by invoking a shmget( ), semget( ), or msgget( ) system call. Just like files, IPC resources are persistent: they must be explicitly deallocated by the creator process, by the current owner, or by a superuser process.

Semaphores are similar to those described in the section "Synchronization and Critical Regions," earlier in this chapter, except that they are reserved for processes in User Mode. Message queues allow processes to exchange messages by using the msgsnd( ) and msgrcv( ) system calls, which insert a message into a specific message queue and extract a message from it, respectively.

The POSIX standard (IEEE Std 1003.1-2001) defines an IPC mechanism based on message queues, which is usually known as *POSIX message queues*. They are similar to the System V IPC's message queues, but they have a much simpler file-based interface to the applications.

Shared memory provides the fastest way for processes to exchange and share data. A process starts by issuing a shmget( ) system call to create a new shared memory having a required size. After obtaining the IPC resource identifier, the process invokes the shmat( ) system call, which returns the starting address of the new region within

the process address space. When the process wishes to detach the shared memory from its address space, it invokes the `shmdt()` system call. The implementation of shared memory depends on how the kernel implements process address spaces.

Process Management

Unix makes a neat distinction between the process and the program it is executing. To that end, the `fork()` and `_exit()` system calls are used respectively to create a new process and to terminate it, while an `exec()`-like system call is invoked to load a new program. After such a system call is executed, the process resumes execution with a brand new address space containing the loaded program.

The process that invokes a `fork()` is the *parent*, while the new process is its *child*. Parents and children can find one another because the data structure describing each process includes a pointer to its immediate parent and pointers to all its immediate children.

A naive implementation of the `fork()` would require both the parent's data and the parent's code to be duplicated and the copies assigned to the child. This would be quite time consuming. Current kernels that can rely on hardware paging units follow the Copy-On-Write approach, which defers page duplication until the last moment (i.e., until the parent or the child is required to write into a page). We shall describe how Linux implements this technique in the section "Copy On Write" in Chapter 9.

The `_exit()` system call terminates a process. The kernel handles this system call by releasing the resources owned by the process and sending the parent process a `SIGCHLD` signal, which is ignored by default.

Zombie processes

How can a parent process inquire about termination of its children? The `wait4()` system call allows a process to wait until one of its children terminates; it returns the process ID (PID) of the terminated child.

When executing this system call, the kernel checks whether a child has already terminated. A special *zombie* process state is introduced to represent terminated processes: a process remains in that state until its parent process executes a `wait4()` system call on it. The system call handler extracts data about resource usage from the process descriptor fields; the process descriptor may be released once the data is collected. If no child process has already terminated when the `wait4()` system call is executed, the kernel usually puts the process in a wait state until a child terminates.

Many kernels also implement a `waitpid()` system call, which allows a process to wait for a specific child process. Other variants of `wait4()` system calls are also quite common.

It's good practice for the kernel to keep around information on a child process until the parent issues its wait4( ) call, but suppose the parent process terminates without issuing that call? The information takes up valuable memory slots that could be used to serve living processes. For example, many shells allow the user to start a command in the background and then log out. The process that is running the command shell terminates, but its children continue their execution.

The solution lies in a special system process called *init*, which is created during system initialization. When a process terminates, the kernel changes the appropriate process descriptor pointers of all the existing children of the terminated process to make them become children of *init*. This process monitors the execution of all its children and routinely issues wait4( ) system calls, whose side effect is to get rid of all orphaned zombies.

Process groups and login sessions

Modern Unix operating systems introduce the notion of *process groups* to represent a "job" abstraction. For example, in order to execute the command line:

```
$ ls | sort | more
```

a shell that supports process groups, such as bash, creates a new group for the three processes corresponding to ls, sort, and more. In this way, the shell acts on the three processes as if they were a single entity (the job, to be precise). Each process descriptor includes a field containing the *process group ID*. Each group of processes may have a *group leader*, which is the process whose PID coincides with the process group ID. A newly created process is initially inserted into the process group of its parent.

Modern Unix kernels also introduce *login sessions*. Informally, a login session contains all processes that are descendants of the process that has started a working session on a specific terminal—usually, the first command shell process created for the user. All processes in a process group must be in the same login session. A login session may have several process groups active simultaneously; one of these process groups is always in the foreground, which means that it has access to the terminal. The other active process groups are in the background. When a background process tries to access the terminal, it receives a SIGTTIN or SIGTTOUT signal. In many command shells, the internal commands bg and fg can be used to put a process group in either the background or the foreground.

Memory Management

Memory management is by far the most complex activity in a Unix kernel. More than a third of this book is dedicated just to describing how Linux handles memory management. This section illustrates some of the main issues related to memory management.

Virtual memory

All recent Unix systems provide a useful abstraction called *virtual memory*. Virtual memory acts as a logical layer between the application memory requests and the hardware Memory Management Unit (MMU). Virtual memory has many purposes and advantages:

- Several processes can be executed concurrently.
- It is possible to run applications whose memory needs are larger than the available physical memory.
- Processes can execute a program whose code is only partially loaded in memory.
- Each process is allowed to access a subset of the available physical memory.
- Processes can share a single memory image of a library or program.
- Programs can be relocatable—that is, they can be placed anywhere in physical memory.
- Programmers can write machine-independent code, because they do not need to be concerned about physical memory organization.

The main ingredient of a virtual memory subsystem is the notion of *virtual address space*. The set of memory references that a process can use is different from physical memory addresses. When a process uses a virtual address,[*] the kernel and the MMU cooperate to find the actual physical location of the requested memory item.

Today's CPUs include hardware circuits that automatically translate the virtual addresses into physical ones. To that end, the available RAM is partitioned into page frames—typically 4 or 8 KB in length—and a set of Page Tables is introduced to specify how virtual addresses correspond to physical addresses. These circuits make memory allocation simpler, because a request for a block of contiguous virtual addresses can be satisfied by allocating a group of page frames having noncontiguous physical addresses.

Random access memory usage

All Unix operating systems clearly distinguish between two portions of the random access memory (RAM). A few megabytes are dedicated to storing the kernel image (i.e., the kernel code and the kernel static data structures). The remaining portion of RAM is usually handled by the virtual memory system and is used in three possible ways:

- To satisfy kernel requests for buffers, descriptors, and other dynamic kernel data structures
- To satisfy process requests for generic memory areas and for memory mapping of files

---

[*] These addresses have different nomenclatures, depending on the computer architecture. As we'll see in Chapter 2, Intel manuals refer to them as "logical addresses."

- To get better performance from disks and other buffered devices by means of caches

Each request type is valuable. On the other hand, because the available RAM is limited, some balancing among request types must be done, particularly when little available memory is left. Moreover, when some critical threshold of available memory is reached and a page-frame-reclaiming algorithm is invoked to free additional memory, which are the page frames most suitable for reclaiming? As we will see in Chapter 17, there is no simple answer to this question and very little support from theory. The only available solution lies in developing carefully tuned empirical algorithms.

One major problem that must be solved by the virtual memory system is *memory fragmentation*. Ideally, a memory request should fail only when the number of free page frames is too small. However, the kernel is often forced to use physically contiguous memory areas. Hence the memory request could fail even if there is enough memory available, but it is not available as one contiguous chunk.

Kernel Memory Allocator

The *Kernel Memory Allocator* (*KMA*) is a subsystem that tries to satisfy the requests for memory areas from all parts of the system. Some of these requests come from other kernel subsystems needing memory for kernel use, and some requests come via system calls from user programs to increase their processes' address spaces. A good KMA should have the following features:

- It must be fast. Actually, this is the most crucial attribute, because it is invoked by all kernel subsystems (including the interrupt handlers).
- It should minimize the amount of wasted memory.
- It should try to reduce the memory fragmentation problem.
- It should be able to cooperate with the other memory management subsystems to borrow and release page frames from them.

Several proposed KMAs, which are based on a variety of different algorithmic techniques, include:

- Resource map allocator
- Power-of-two free lists
- McKusick-Karels allocator
- Buddy system
- Mach's Zone allocator
- Dynix allocator
- Solaris's Slab allocator

As we will see in Chapter 8, Linux's KMA uses a Slab allocator on top of a buddy system.

Process virtual address space handling

The address space of a process contains all the virtual memory addresses that the process is allowed to reference. The kernel usually stores a process virtual address space as a list of *memory area descriptors*. For example, when a process starts the execution of some program via an exec( )-like system call, the kernel assigns to the process a virtual address space that comprises memory areas for:

- The executable code of the program
- The initialized data of the program
- The uninitialized data of the program
- The initial program stack (i.e., the User Mode stack)
- The executable code and data of needed shared libraries
- The heap (the memory dynamically requested by the program)

All recent Unix operating systems adopt a memory allocation strategy called *demand paging*. With demand paging, a process can start program execution with none of its pages in physical memory. As it accesses a nonpresent page, the MMU generates an exception; the exception handler finds the affected memory region, allocates a free page, and initializes it with the appropriate data. In a similar fashion, when the process dynamically requires memory by using malloc( ), or the brk( ) system call (which is invoked internally by malloc( )), the kernel just updates the size of the heap memory region of the process. A page frame is assigned to the process only when it generates an exception by trying to refer its virtual memory addresses.

Virtual address spaces also allow other efficient strategies, such as the Copy On Write strategy mentioned earlier. For example, when a new process is created, the kernel just assigns the parent's page frames to the child address space, but marks them read-only. An exception is raised as soon as the parent or the child tries to modify the contents of a page. The exception handler assigns a new page frame to the affected process and initializes it with the contents of the original page.

Caching

A good part of the available physical memory is used as cache for hard disks and other block devices. This is because hard drives are very slow: a disk access requires several milliseconds, which is a very long time compared with the RAM access time. Therefore, disks are often the bottleneck in system performance. As a general rule, one of the policies already implemented in the earliest Unix system is to defer writing to disk as long as possible. As a result, data read previously from disk and no longer used by any process continue to stay in RAM.

This strategy is based on the fact that there is a good chance that new processes will require data read from or written to disk by processes that no longer exist. When a process asks to access a disk, the kernel checks first whether the required data are in the cache. Each time this happens (a cache hit), the kernel is able to service the process request without accessing the disk.

The sync( ) system call forces disk synchronization by writing all of the "dirty" buffers (i.e., all the buffers whose contents differ from that of the corresponding disk blocks) into disk. To avoid data loss, all operating systems take care to periodically write dirty buffers back to disk.

Device Drivers

The kernel interacts with I/O devices by means of *device drivers*. Device drivers are included in the kernel and consist of data structures and functions that control one or more devices, such as hard disks, keyboards, mouses, monitors, network interfaces, and devices connected to an SCSI bus. Each driver interacts with the remaining part of the kernel (even with other drivers) through a specific interface. This approach has the following advantages:

- Device-specific code can be encapsulated in a specific module.
- Vendors can add new devices without knowing the kernel source code; only the interface specifications must be known.
- The kernel deals with all devices in a uniform way and accesses them through the same interface.
- It is possible to write a device driver as a module that can be dynamically loaded in the kernel without requiring the system to be rebooted. It is also possible to dynamically unload a module that is no longer needed, therefore minimizing the size of the kernel image stored in RAM.

Figure 1-4 illustrates how device drivers interface with the rest of the kernel and with the processes.
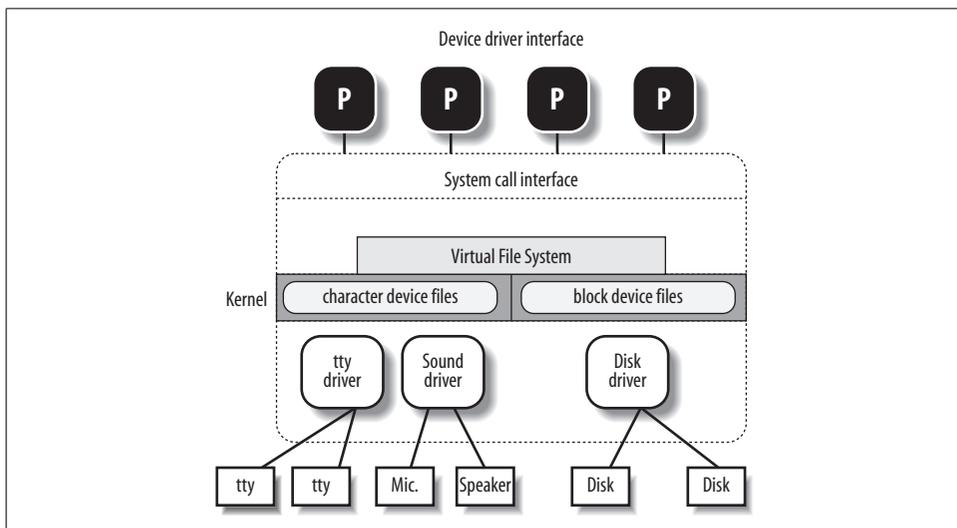
Figure 1-4. Device driver interface

Some user programs (P) wish to operate on hardware devices. They make requests to the kernel using the usual file-related system calls and the device files normally found in the */dev* directory. Actually, the device files are the user-visible portion of the device driver interface. Each device file refers to a specific device driver, which is invoked by the kernel to perform the requested operation on the hardware component.

At the time Unix was introduced, graphical terminals were uncommon and expensive, so only alphanumeric terminals were handled directly by Unix kernels. When graphical terminals became widespread, ad hoc applications such as the X Window System were introduced that ran as standard processes and accessed the I/O ports of the graphics interface and the RAM video area directly. Some recent Unix kernels, such as Linux 2.6, provide an abstraction for the frame buffer of the graphic card and allow application software to access them without needing to know anything about the I/O ports of the graphics interface (see the section "Levels of Kernel Support" in Chapter 13.)

Memory Addressing

This chapter deals with addressing techniques. Luckily, an operating system is not forced to keep track of physical memory all by itself; today's microprocessors include several hardware circuits to make memory management both more efficient and more robust so that programming errors cannot cause improper accesses to memory outside the program.

As in the rest of this book, we offer details in this chapter on how 80×86 microprocessors address memory chips and how Linux uses the available addressing circuits. You will find, we hope, that when you learn the implementation details on Linux's most popular platform you will better understand both the general theory of paging and how to research the implementation on other platforms.

This is the first of three chapters related to memory management; Chapter 8 discusses how the kernel allocates main memory to itself, while Chapter 9 considers how linear addresses are assigned to processes.

Memory Addresses

Programmers casually refer to a *memory address* as the way to access the contents of a memory cell. But when dealing with 80×86 microprocessors, we have to distinguish three kinds of addresses:

Logical address
> Included in the machine language instructions to specify the address of an operand or of an instruction. This type of address embodies the well-known 80×86 segmented architecture that forces MS-DOS and Windows programmers to divide their programs into segments. Each logical address consists of a *segment* and an *offset* (or *displacement*) that denotes the distance from the start of the segment to the actual address.

Linear address (also known as virtual address)
> A single 32-bit unsigned integer that can be used to address up to 4 GB—that is, up to 4,294,967,296 memory cells. Linear addresses are usually represented in hexadecimal notation; their values range from 0x00000000 to 0xffffffff.

Physical address
> Used to address memory cells in memory chips. They correspond to the electrical signals sent along the address pins of the microprocessor to the memory bus. Physical addresses are represented as 32-bit or 36-bit unsigned integers.

The Memory Management Unit (MMU) transforms a logical address into a linear address by means of a hardware circuit called a segmentation unit; subsequently, a second hardware circuit called a paging unit transforms the linear address into a physical address (see Figure 2-1).

Figure 2-1. Logical address translation

In multiprocessor systems, all CPUs usually share the same memory; this means that RAM chips may be accessed concurrently by independent CPUs. Because read or write operations on a RAM chip must be performed serially, a hardware circuit called a *memory arbiter* is inserted between the bus and every RAM chip. Its role is to grant access to a CPU if the chip is free and to delay it if the chip is busy servicing a request by another processor. Even uniprocessor systems use memory arbiters, because they include specialized processors called *DMA controllers* that operate concurrently with the CPU (see the section "Direct Memory Access (DMA)" in Chapter 13). In the case of multiprocessor systems, the structure of the arbiter is more complex because it has more input ports. The dual Pentium, for instance, maintains a two-port arbiter at each chip entrance and requires that the two CPUs exchange synchronization messages before attempting to use the common bus. From the programming point of view, the arbiter is hidden because it is managed by hardware circuits.

Segmentation in Hardware

Starting with the 80286 model, Intel microprocessors perform address translation in two different ways called *real mode* and *protected mode*. We'll focus in the next sections on address translation when protected mode is enabled. Real mode exists mostly to maintain processor compatibility with older models and to allow the operating system to bootstrap (see Appendix A for a short description of real mode).

Segment Selectors and Segmentation Registers

A logical address consists of two parts: a segment identifier and an offset that specifies the relative address within the segment. The segment identifier is a 16-bit field called the *Segment Selector* (see Figure 2-2), while the offset is a 32-bit field. We'll describe the fields of Segment Selectors in the section "Fast Access to Segment Descriptors" later in this chapter.
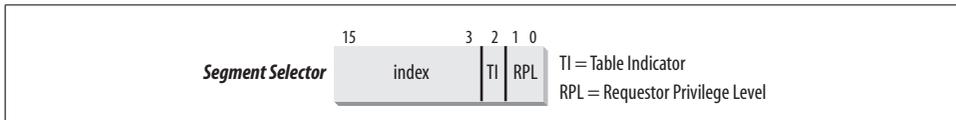

Figure 2-2. Segment Selector format

To make it easy to retrieve segment selectors quickly, the processor provides *segmentation registers* whose only purpose is to hold Segment Selectors; these registers are called cs, ss, ds, es, fs, and gs. Although there are only six of them, a program can reuse the same segmentation register for different purposes by saving its content in memory and then restoring it later.

Three of the six segmentation registers have specific purposes:

cs The code segment register, which points to a segment containing program instructions

ss The stack segment register, which points to a segment containing the current program stack

ds The data segment register, which points to a segment containing global and static data

The remaining three segmentation registers are general purpose and may refer to arbitrary data segments.

The cs register has another important function: it includes a 2-bit field that specifies the Current Privilege Level (CPL) of the CPU. The value 0 denotes the highest privilege level, while the value 3 denotes the lowest one. Linux uses only levels 0 and 3, which are respectively called Kernel Mode and User Mode.

Segment Descriptors

Each segment is represented by an 8-byte *Segment Descriptor* that describes the segment characteristics. Segment Descriptors are stored either in the *Global Descriptor Table* (*GDT*) or in the *Local Descriptor Table* (*LDT*).

Usually only one GDT is defined, while each process is permitted to have its own LDT if it needs to create additional segments besides those stored in the GDT. The address and size of the GDT in main memory are contained in the gdtr control register, while the address and size of the currently used LDT are contained in the ldtr control register.

Figure 2-3 illustrates the format of a Segment Descriptor; the meaning of the various fields is explained in Table 2-1.

Table 2-1. Segment Descriptor fields

Field name	Description
Base	Contains the linear address of the first byte of the segment.
G	*Granularity flag*: if it is cleared (equal to 0), the segment size is expressed in bytes; otherwise, it is expressed in multiples of 4096 bytes.
Limit	Holds the offset of the last memory cell in the segment, thus binding the segment length. When G is set to 0, the size of a segment may vary between 1 byte and 1 MB; otherwise, it may vary between 4 KB and 4 GB.
S	*System flag*: if it is cleared, the segment is a *system segment* that stores critical data structures such as the Local Descriptor Table; otherwise, it is a normal code or data segment.
Type	Characterizes the segment type and its access rights (see the text that follows this table).
DPL	*Descriptor Privilege Level*: used to restrict accesses to the segment. It represents the minimal CPU privilege level requested for accessing the segment. Therefore, a segment with its DPL set to 0 is accessible only when the CPL is 0 — that is, in Kernel Mode — while a segment with its DPL set to 3 is accessible with every CPL value.
P	*Segment-Present flag*: is equal to 0 if the segment is not stored currently in main memory. Linux always sets this flag (bit 47) to 1, because it never swaps out whole segments to disk.
D or B	Called D or B depending on whether the segment contains code or data. Its meaning is slightly different in the two cases, but it is basically set (equal to 1) if the addresses used as segment offsets are 32 bits long, and it is cleared if they are 16 bits long (see the Intel manual for further details).
AVL	May be used by the operating system, but it is ignored by Linux.

There are several types of segments, and thus several types of Segment Descriptors. The following list shows the types that are widely used in Linux.

Code Segment Descriptor
Indicates that the Segment Descriptor refers to a code segment; it may be included either in the GDT or in the LDT. The descriptor has the S flag set (non-system segment).

Data Segment Descriptor
Indicates that the Segment Descriptor refers to a data segment; it may be included either in the GDT or in the LDT. The descriptor has the S flag set. Stack segments are implemented by means of generic data segments.

Task State Segment Descriptor (TSSD)
Indicates that the Segment Descriptor refers to a Task State Segment (TSS)— that is, a segment used to save the contents of the processor registers (see the section "Task State Segment" in Chapter 3); it can appear only in the GDT. The corresponding Type field has the value 11 or 9, depending on whether the corresponding process is currently executing on a CPU. The S flag of such descriptors is set to 0.
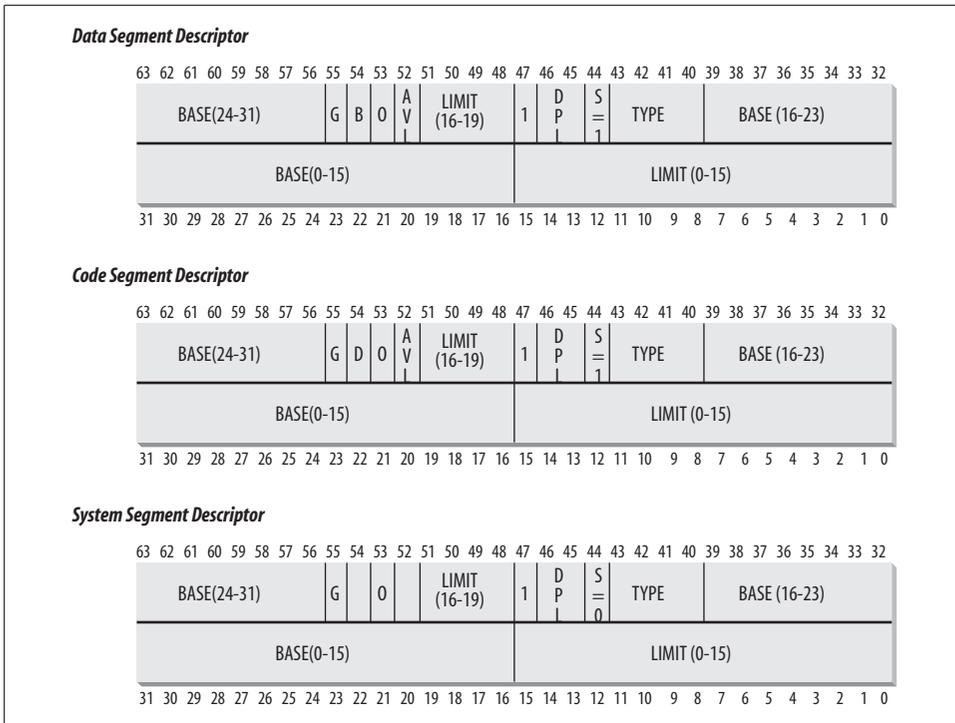
Data Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

| BASE(24-31) | G | B | 0 | AVL | LIMIT (16-19) | 1 | DPL | S = 1 | TYPE | BASE (16-23) |
| BASE(0-15) | | | | | | LIMIT (0-15) | | | | |

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Code Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

| BASE(24-31) | G | D | 0 | AVL | LIMIT (16-19) | 1 | DPL | S = 1 | TYPE | BASE (16-23) |
| BASE(0-15) | | | | | | LIMIT (0-15) | | | | |

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

System Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

| BASE(24-31) | G | 0 | LIMIT (16-19) | 1 | DPL | S = 0 | TYPE | BASE (16-23) |
| BASE(0-15) | | | LIMIT (0-15) | | | | | |

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Figure 2-3. Segment Descriptor format

Local Descriptor Table Descriptor (LDTD)

> Indicates that the Segment Descriptor refers to a segment containing an LDT; it can appear only in the GDT. The corresponding Type field has the value 2. The S flag of such descriptors is set to 0. The next section shows how 80×86 processors are able to decide whether a segment descriptor is stored in the GDT or in the LDT of the process.

Fast Access to Segment Descriptors

We recall that logical addresses consist of a 16-bit Segment Selector and a 32-bit Offset, and that segmentation registers store only the Segment Selector.

To speed up the translation of logical addresses into linear addresses, the 80×86 processor provides an additional nonprogrammable register—that is, a register that cannot be set by a programmer—for each of the six programmable segmentation registers. Each nonprogrammable register contains the 8-byte Segment Descriptor (described in the previous section) specified by the Segment Selector contained in the corresponding segmentation register. Every time a Segment Selector is loaded in a segmentation register, the corresponding Segment Descriptor is loaded from memory into the matching nonprogrammable CPU register. From then on, translations of logical addresses referring to that segment can be performed without accessing the GDT

or LDT stored in main memory; the processor can refer only directly to the CPU register containing the Segment Descriptor. Accesses to the GDT or LDT are necessary only when the contents of the segmentation registers change (see Figure 2-4).
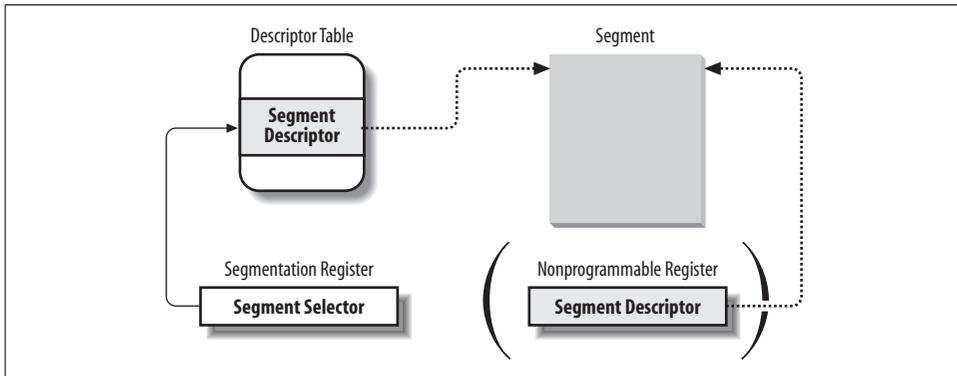
Figure 2-4. Segment Selector and Segment Descriptor

Any Segment Selector includes three fields that are described in Table 2-2.

Table 2-2. Segment Selector fields

Field name	Description
index	Identifies the Segment Descriptor entry contained in the GDT or in the LDT (described further in the text following this table).
TI	*Table Indicator*: specifies whether the Segment Descriptor is included in the GDT (TI = 0) or in the LDT (TI = 1).
RPL	*Requestor Privilege Level*: specifies the Current Privilege Level of the CPU when the corresponding Segment Selector is loaded into the cs register; it also may be used to selectively weaken the processor privilege level when accessing data segments (see Intel documentation for details).

Because a Segment Descriptor is 8 bytes long, its relative address inside the GDT or the LDT is obtained by multiplying the 13-bit index field of the Segment Selector by 8. For instance, if the GDT is at 0x00020000 (the value stored in the gdtr register) and the index specified by the Segment Selector is 2, the address of the corresponding Segment Descriptor is 0x00020000 + (2 × 8), or 0x00020010.

The first entry of the GDT is always set to 0. This ensures that logical addresses with a null Segment Selector will be considered invalid, thus causing a processor exception. The maximum number of Segment Descriptors that can be stored in the GDT is 8,191 (i.e., $2^{13}-1$).

Segmentation Unit

Figure 2-5 shows in detail how a logical address is translated into a corresponding linear address. The *segmentation unit* performs the following operations:

- Examines the `TI` field of the Segment Selector to determine which Descriptor Table stores the Segment Descriptor. This field indicates that the Descriptor is either in the GDT (in which case the segmentation unit gets the base linear address of the GDT from the `gdtr` register) or in the active LDT (in which case the segmentation unit gets the base linear address of that LDT from the `ldtr` register).
- Computes the address of the Segment Descriptor from the `index` field of the Segment Selector. The `index` field is multiplied by 8 (the size of a Segment Descriptor), and the result is added to the content of the `gdtr` or `ldtr` register.
- Adds the offset of the logical address to the `Base` field of the Segment Descriptor, thus obtaining the linear address.
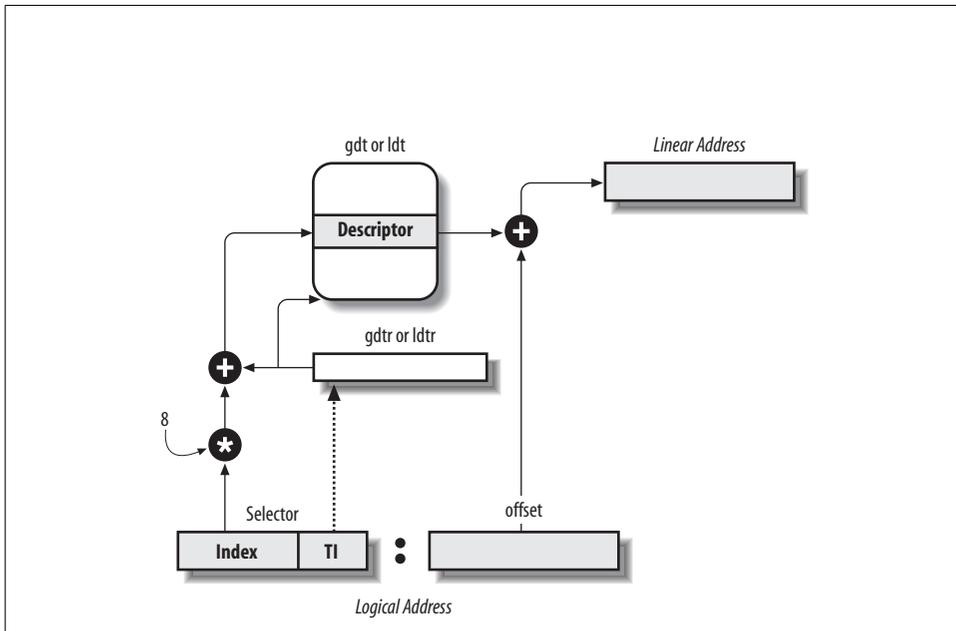
Figure 2-5. Translating a logical address

Notice that, thanks to the nonprogrammable registers associated with the segmentation registers, the first two operations need to be performed only when a segmentation register has been changed.

Segmentation in Linux

Segmentation has been included in 80×86 microprocessors to encourage programmers to split their applications into logically related entities, such as subroutines or global and local data areas. However, Linux uses segmentation in a very limited way. In fact, segmentation and paging are somewhat redundant, because both can be used

to separate the physical address spaces of processes: segmentation can assign a differ-ent linear address space to each process, while paging can map the same linear address space into different physical address spaces. Linux prefers paging to segmentation for the following reasons:

- Memory management is simpler when all processes use the same segment regis-ter values—that is, when they share the same set of linear addresses.
- One of the design objectives of Linux is portability to a wide range of architec-tures; RISC architectures in particular have limited support for segmentation.

The 2.6 version of Linux uses segmentation only when required by the 80×86 archi-tecture.

All Linux processes running in User Mode use the same pair of segments to address instructions and data. These segments are called *user code segment* and *user data seg-ment*, respectively. Similarly, all Linux processes running in Kernel Mode use the same pair of segments to address instructions and data: they are called *kernel code segment* and *kernel data segment*, respectively. Table 2-3 shows the values of the Seg-ment Descriptor fields for these four crucial segments.

Table 2-3. Values of the Segment Descriptor fields for the four main Linux segments

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

The corresponding Segment Selectors are defined by the macros __USER_CS, __USER_DS, __KERNEL_CS, and __KERNEL_DS, respectively. To address the kernel code segment, for instance, the kernel just loads the value yielded by the __KERNEL_CS macro into the cs segmentation register.

Notice that the linear addresses associated with such segments all start at 0 and reach the addressing limit of $2^{32} -1$. This means that all processes, either in User Mode or in Kernel Mode, may use the same logical addresses.

Another important consequence of having all segments start at 0x00000000 is that in Linux, logical addresses coincide with linear addresses; that is, the value of the Off-set field of a logical address always coincides with the value of the corresponding lin-ear address.

As stated earlier, the Current Privilege Level of the CPU indicates whether the proces-sor is in User or Kernel Mode and is specified by the RPL field of the Segment Selector stored in the cs register. Whenever the CPL is changed, some segmentation registers must be correspondingly updated. For instance, when the CPL is equal to 3 (User Mode), the ds register must contain the Segment Selector of the user data segment,

but when the CPL is equal to 0, the `ds` register must contain the Segment Selector of the kernel data segment.

A similar situation occurs for the `ss` register. It must refer to a User Mode stack inside the user data segment when the CPL is 3, and it must refer to a Kernel Mode stack inside the kernel data segment when the CPL is 0. When switching from User Mode to Kernel Mode, Linux always makes sure that the `ss` register contains the Segment Selector of the kernel data segment.

When saving a pointer to an instruction or to a data structure, the kernel does not need to store the Segment Selector component of the logical address, because the `ss` register contains the current Segment Selector. As an example, when the kernel invokes a function, it executes a `call` assembly language instruction specifying just the Offset component of its logical address; the Segment Selector is implicitly selected as the one referred to by the `cs` register. Because there is just one segment of type "executable in Kernel Mode," namely the code segment identified by `__KERNEL_CS`, it is sufficient to load `__KERNEL_CS` into `cs` whenever the CPU switches to Kernel Mode. The same argument goes for pointers to kernel data structures (implicitly using the `ds` register), as well as for pointers to user data structures (the kernel explicitly uses the `es` register).

Besides the four segments just described, Linux makes use of a few other specialized segments. We'll introduce them in the next section while describing the Linux GDT.

The Linux GDT

In uniprocessor systems there is only one GDT, while in multiprocessor systems there is one GDT for every CPU in the system. All GDTs are stored in the `cpu_gdt_table` array, while the addresses and sizes of the GDTs (used when initializing the `gdtr` registers) are stored in the `cpu_gdt_descr` array. If you look in the Source Code Index, you can see that these symbols are defined in the file *arch/i386/kernel/head.S*. Every macro, function, and other symbol in this book is listed in the Source Code Index, so you can quickly find it in the source code.

The layout of the GDTs is shown schematically in Figure 2-6. Each GDT includes 18 segment descriptors and 14 null, unused, or reserved entries. Unused entries are inserted on purpose so that Segment Descriptors usually accessed together are kept in the same 32-byte line of the hardware cache (see the section "Hardware Cache" later in this chapter).

The 18 segment descriptors included in each GDT point to the following segments:

- Four user and kernel code and data segments (see previous section).
- A Task State Segment (TSS), different for each processor in the system. The linear address space corresponding to a TSS is a small subset of the linear address space corresponding to the kernel data segment. The Task State Segments are

Linux's GDT	Segment Selectors		Linux's GDT	Segment Selectors
null	0x0		TSS	0x80
reserved			LDT	0x88
reserved			PNPBIOS 32-bit code	0x90
reserved			PNPBIOS 16-bit code	0x98
not used			PNPBIOS 16-bit data	0xa0
not used			PNPBIOS 16-bit data	0xa8
TLS #1	0x33		PNPBIOS 16-bit data	0xb0
TLS #2	0x3b		APMBIOS 32-bit code	0xb8
TLS #3	0x43		APMBIOS 16-bit code	0xc0
reserved			APMBIOS data	0xc8
reserved			not used	
reserved			not used	
kernel code	0x60 (__KERNEL_CS)		not used	
kernel data	0x68 (__KERNEL_DS)		not used	
user code	0x73 (__USER_CS)		not used	
user data	0x7b (__USER_DS)		double fault TSS	0xf8

Figure 2-6. The Global Descriptor Table

sequentially stored in the `init_tss` array; in particular, the `Base` field of the TSS descriptor for the *n*th CPU points to the *n*th component of the `init_tss` array. The `G` (granularity) flag is cleared, while the `Limit` field is set to `0xeb`, because the TSS segment is 236 bytes long. The `Type` field is set to 9 or 11 (available 32-bit TSS), and the `DPL` is set to 0, because processes in User Mode are not allowed to access TSS segments. You will find details on how Linux uses TSSs in the section "Task State Segment" in Chapter 3.

- A segment including the default Local Descriptor Table (LDT), usually shared by all processes (see the next section).

- Three *Thread-Local Storage (TLS)* segments: this is a mechanism that allows multithreaded applications to make use of up to three segments containing data local to each thread. The `set_thread_area()` and `get_thread_area()` system calls, respectively, create and release a TLS segment for the executing process.

- Three segments related to Advanced Power Management (APM): the BIOS code makes use of segments, so when the Linux APM driver invokes BIOS functions to get or set the status of APM devices, it may use custom code and data segments.

- Five segments related to Plug and Play (PnP) BIOS services. As in the previous case, the BIOS code makes use of segments, so when the Linux PnP driver invokes BIOS functions to detect the resources used by PnP devices, it may use custom code and data segments.

- A special TSS segment used by the kernel to handle "Double fault" exceptions (see "Exceptions" in Chapter 4).

As stated earlier, there is a copy of the GDT for each processor in the system. All copies of the GDT store identical entries, except for a few cases. First, each processor has its own TSS segment, thus the corresponding GDT's entries differ. Moreover, a few entries in the GDT may depend on the process that the CPU is executing (LDT and TLS Segment Descriptors). Finally, in some cases a processor may temporarily modify an entry in its copy of the GDT; this happens, for instance, when invoking an APM's BIOS procedure.

The Linux LDTs

Most Linux User Mode applications do not make use of a Local Descriptor Table, thus the kernel defines a default LDT to be shared by most processes. The default Local Descriptor Table is stored in the `default_ldt` array. It includes five entries, but only two of them are effectively used by the kernel: a call gate for iBCS executables, and a call gate for Solaris/x86 executables (see the section "Execution Domains" in Chapter 20). *Call gates* are a mechanism provided by 80 × 86 microprocessors to change the privilege level of the CPU while invoking a predefined function; as we won't discuss them further, you should consult the Intel documentation for more details.

In some cases, however, processes may require to set up their own LDT. This turns out to be useful to applications (such as Wine) that execute segment-oriented Microsoft Windows applications. The `modify_ldt()` system call allows a process to do this.

Any custom LDT created by `modify_ldt()` also requires its own segment. When a processor starts executing a process having a custom LDT, the LDT entry in the CPU-specific copy of the GDT is changed accordingly.

User Mode applications also may allocate new segments by means of `modify_ldt()`; the kernel, however, never makes use of these segments, and it does not have to keep track of the corresponding Segment Descriptors, because they are included in the custom LDT of the process.

Paging in Hardware

The paging unit translates linear addresses into physical ones. One key task in the unit is to check the requested access type against the access rights of the linear address. If the memory access is not valid, it generates a Page Fault exception (see Chapter 4 and Chapter 8).

For the sake of efficiency, linear addresses are grouped in fixed-length intervals called *pages*; contiguous linear addresses within a page are mapped into contiguous physical addresses. In this way, the kernel can specify the physical address and the access rights of a page instead of those of all the linear addresses included in it. Following the usual convention, we shall use the term "page" to refer both to a set of linear addresses and to the data contained in this group of addresses.

The paging unit thinks of all RAM as partitioned into fixed-length *page frames* (sometimes referred to as *physical pages*). Each page frame contains a page—that is, the length of a page frame coincides with that of a page. A page frame is a constituent of main memory, and hence it is a storage area. It is important to distinguish a page from a page frame; the former is just a block of data, which may be stored in any page frame or on disk.

The data structures that map linear to physical addresses are called *page tables*; they are stored in main memory and must be properly initialized by the kernel before enabling the paging unit.

Starting with the 80386, all 80×86 processors support paging; it is enabled by setting the PG flag of a control register named cr0. When PG = 0, linear addresses are interpreted as physical addresses.

Regular Paging

Starting with the 80386, the paging unit of Intel processors handles 4 KB pages.

The 32 bits of a linear address are divided into three fields:

Directory
    The most significant 10 bits

Table
    The intermediate 10 bits

Offset
    The least significant 12 bits

The translation of linear addresses is accomplished in two steps, each based on a type of translation table. The first translation table is called the *Page Directory*, and the second is called the *Page Table*.[*]

The aim of this two-level scheme is to reduce the amount of RAM required for per-process Page Tables. If a simple one-level Page Table was used, then it would require

---

[*] In the discussion that follows, the lowercase "page table" term denotes any page storing the mapping between linear and physical addresses, while the capitalized "Page Table" term denotes a page in the last level of page tables.

up to 2^{20} entries (i.e., at 4 bytes per entry, 4 MB of RAM) to represent the Page Table for each process (if the process used a full 4 GB linear address space), even though a process does not use all addresses in that range. The two-level scheme reduces the memory by requiring Page Tables only for those virtual memory regions actually used by a process.

Each active process must have a Page Directory assigned to it. However, there is no need to allocate RAM for all Page Tables of a process at once; it is more efficient to allocate RAM for a Page Table only when the process effectively needs it.

The physical address of the Page Directory in use is stored in a control register named cr3. The Directory field within the linear address determines the entry in the Page Directory that points to the proper Page Table. The address's Table field, in turn, determines the entry in the Page Table that contains the physical address of the page frame containing the page. The Offset field determines the relative position within the page frame (see Figure 2-7). Because it is 12 bits long, each page consists of 4096 bytes of data.
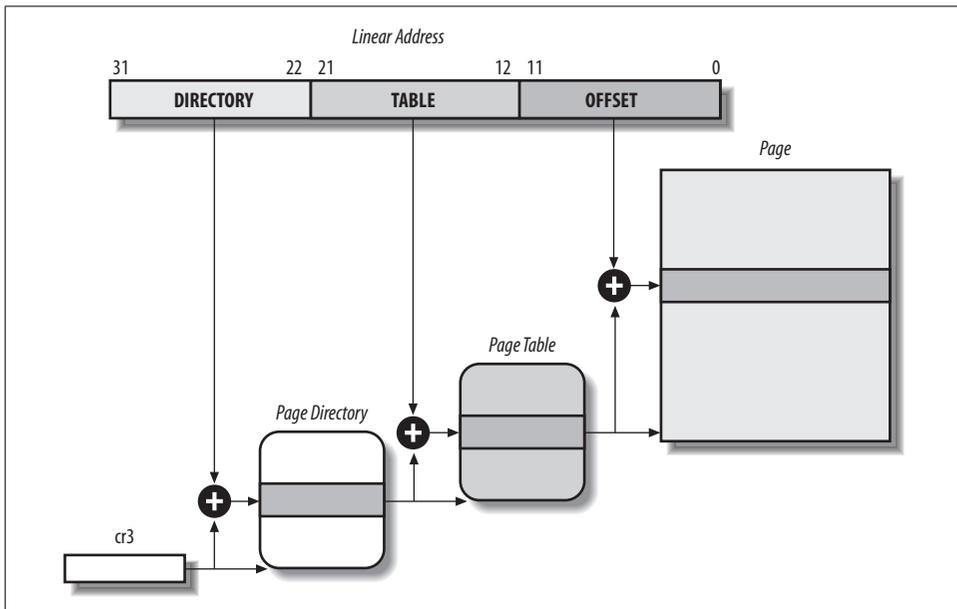
Figure 2-7. Paging by 80×86 processors

Both the Directory and the Table fields are 10 bits long, so Page Directories and Page Tables can include up to 1,024 entries. It follows that a Page Directory can address up to $1024 \times 1024 \times 4096 = 2^{32}$ memory cells, as you'd expect in 32-bit addresses.

The entries of Page Directories and Page Tables have the same structure. Each entry includes the following fields:

Present *flag*

> If it is set, the referred-to page (or Page Table) is contained in main memory; if the flag is 0, the page is not contained in main memory and the remaining entry bits may be used by the operating system for its own purposes. If the entry of a Page Table or Page Directory needed to perform an address translation has the Present flag cleared, the paging unit stores the linear address in a control register named cr2 and generates exception 14: the Page Fault exception. (We will see in Chapter 17 how Linux uses this field.)

Field containing the 20 most significant bits of a page frame physical address

> Because each page frame has a 4-KB capacity, its physical address must be a multiple of 4096, so the 12 least significant bits of the physical address are always equal to 0. If the field refers to a Page Directory, the page frame contains a Page Table; if it refers to a Page Table, the page frame contains a page of data.

Accessed *flag*

> Set each time the paging unit addresses the corresponding page frame. This flag may be used by the operating system when selecting pages to be swapped out. The paging unit never resets this flag; this must be done by the operating system.

Dirty *flag*

> Applies only to the Page Table entries. It is set each time a write operation is performed on the page frame. As with the Accessed flag, Dirty may be used by the operating system when selecting pages to be swapped out. The paging unit never resets this flag; this must be done by the operating system.

Read/Write *flag*

> Contains the access right (Read/Write or Read) of the page or of the Page Table (see the section "Hardware Protection Scheme" later in this chapter).

User/Supervisor *flag*

> Contains the privilege level required to access the page or Page Table (see the later section "Hardware Protection Scheme").

PCD *and* PWT *flags*

> Controls the way the page or Page Table is handled by the hardware cache (see the section "Hardware Cache" later in this chapter).

Page Size *flag*

> Applies only to Page Directory entries. If it is set, the entry refers to a 2 MB– or 4 MB–long page frame (see the following sections).

Global *flag*

> Applies only to Page Table entries. This flag was introduced in the Pentium Pro to prevent frequently used pages from being flushed from the TLB cache (see the section "Translation Lookaside Buffers (TLB)" later in this chapter). It works only if the Page Global Enable (PGE) flag of register cr4 is set.

Extended Paging

Starting with the Pentium model, 80×86 microprocessors introduce *extended paging*, which allows page frames to be 4 MB instead of 4 KB in size (see Figure 2-8). Extended paging is used to translate large contiguous linear address ranges into corresponding physical ones; in these cases, the kernel can do without intermediate Page Tables and thus save memory and preserve TLB entries (see the section "Translation Lookaside Buffers (TLB)").
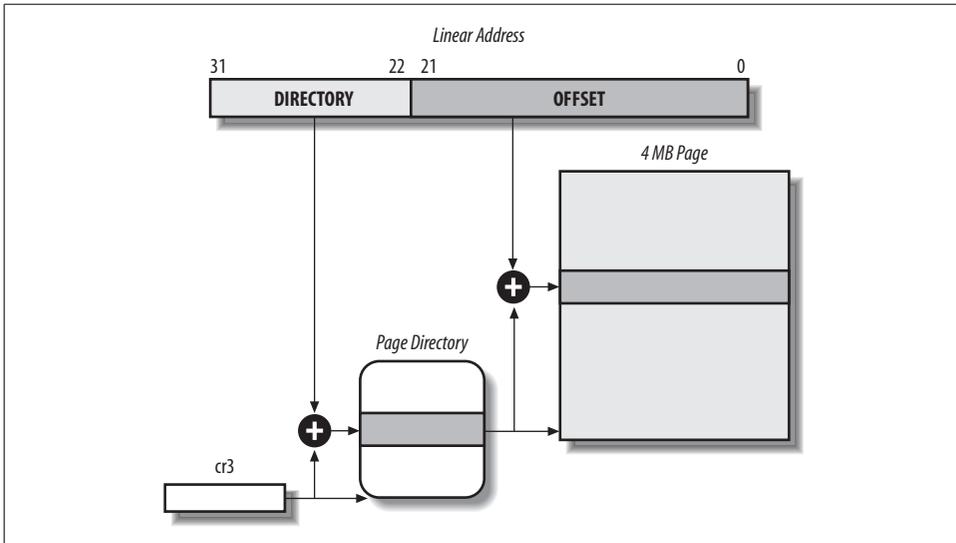
Figure 2-8. Extended paging

As mentioned in the previous section, extended paging is enabled by setting the Page Size flag of a Page Directory entry. In this case, the paging unit divides the 32 bits of a linear address into two fields:

Directory
    The most significant 10 bits

Offset
    The remaining 22 bits

Page Directory entries for extended paging are the same as for normal paging, except that:

* The Page Size flag must be set.
* Only the 10 most significant bits of the 20-bit physical address field are significant. This is because each physical address is aligned on a 4-MB boundary, so the 22 least significant bits of the address are 0.

Extended paging coexists with regular paging; it is enabled by setting the PSE flag of the cr4 processor register.

Hardware Protection Scheme

The paging unit uses a different protection scheme from the segmentation unit. While 80×86 processors allow four possible privilege levels to a segment, only two privilege levels are associated with pages and Page Tables, because privileges are controlled by the User/Supervisor flag mentioned in the earlier section "Regular Paging." When this flag is 0, the page can be addressed only when the CPL is less than 3 (this means, for Linux, when the processor is in Kernel Mode). When the flag is 1, the page can always be addressed.

Furthermore, instead of the three types of access rights (Read, Write, and Execute) associated with segments, only two types of access rights (Read and Write) are associated with pages. If the Read/Write flag of a Page Directory or Page Table entry is equal to 0, the corresponding Page Table or page can only be read; otherwise it can be read and written.[*]

An Example of Regular Paging

A simple example will help in clarifying how regular paging works. Let's assume that the kernel assigns the linear address space between 0x20000000 and 0x2003ffff to a running process.[†] This space consists of exactly 64 pages. We don't care about the physical addresses of the page frames containing the pages; in fact, some of them might not even be in main memory. We are interested only in the remaining fields of the Page Table entries.

Let's start with the 10 most significant bits of the linear addresses assigned to the process, which are interpreted as the Directory field by the paging unit. The addresses start with a 2 followed by zeros, so the 10 bits all have the same value, namely 0x080 or 128 decimal. Thus the Directory field in all the addresses refers to the 129th entry of the process Page Directory. The corresponding entry must contain the physical address of the Page Table assigned to the process (see Figure 2-9). If no other linear addresses are assigned to the process, all the remaining 1,023 entries of the Page Directory are filled with zeros.

---

[*] Recent Intel Pentium 4 processors sport an NX (No eXecute) flag in each 64-bit Page Table entry (PAE must be enabled, see the section "The Physical Address Extension (PAE) Paging Mechanism" later in this chapter). Linux 2.6.11 supports this hardware feature.

[†] As we shall see in the following chapters, the 3 GB linear address space is an upper limit, but a User Mode process is allowed to reference only a subset of it.
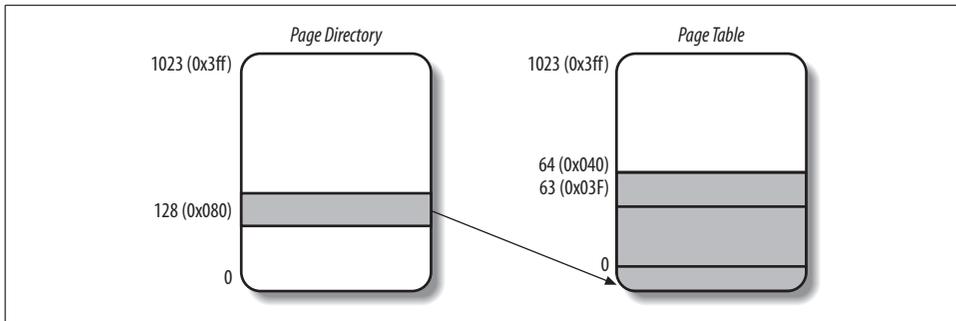
Figure 2-9. An example of paging

The values assumed by the intermediate 10 bits, (that is, the values of the Table field) range from 0 to 0x03f, or from 0 to 63 decimal. Thus, only the first 64 entries of the Page Table are valid. The remaining 960 entries are filled with zeros.

Suppose that the process needs to read the byte at linear address 0x20021406. This address is handled by the paging unit as follows:

1. The Directory field 0x80 is used to select entry 0x80 of the Page Directory, which points to the Page Table associated with the process's pages.

2. The Table field 0x21 is used to select entry 0x21 of the Page Table, which points to the page frame containing the desired page.

3. Finally, the Offset field 0x406 is used to select the byte at offset 0x406 in the desired page frame.

If the Present flag of the 0x21 entry of the Page Table is cleared, the page is not present in main memory; in this case, the paging unit issues a Page Fault exception while translating the linear address. The same exception is issued whenever the process attempts to access linear addresses outside of the interval delimited by 0x20000000 and 0x2003ffff, because the Page Table entries not assigned to the process are filled with zeros; in particular, their Present flags are all cleared.

The Physical Address Extension (PAE) Paging Mechanism

The amount of RAM supported by a processor is limited by the number of address pins connected to the address bus. Older Intel processors from the 80386 to the Pentium used 32-bit physical addresses. In theory, up to 4 GB of RAM could be installed on such systems; in practice, due to the linear address space requirements of User Mode processes, the kernel cannot directly address more than 1 GB of RAM, as we will see in the later section "Paging in Linux."

However, big servers that need to run hundreds or thousands of processes at the same time require more than 4 GB of RAM, and in recent years this created a pressure on Intel to expand the amount of RAM supported on the 32-bit 80×86 architecture.

Intel has satisfied these requests by increasing the number of address pins on its processors from 32 to 36. Starting with the Pentium Pro, all Intel processors are now able to address up to 2^{36} = 64 GB of RAM. However, the increased range of physical addresses can be exploited only by introducing a new paging mechanism that translates 32-bit linear addresses into 36-bit physical ones.

With the Pentium Pro processor, Intel introduced a mechanism called *Physical Address Extension* (*PAE*). Another mechanism, Page Size Extension (PSE-36), was introduced in the Pentium III processor, but Linux does not use it, and we won't discuss it further in this book.

PAE is activated by setting the Physical Address Extension (PAE) flag in the cr4 control register. The Page Size (PS) flag in the page directory entry enables large page sizes (2 MB when PAE is enabled).

Intel has changed the paging mechanism in order to support PAE.

- The 64 GB of RAM are split into 2^{24} distinct page frames, and the physical address field of Page Table entries has been expanded from 20 to 24 bits. Because a PAE Page Table entry must include the 12 flag bits (described in the earlier section "Regular Paging") and the 24 physical address bits, for a grand total of 36, the Page Table entry size has been doubled from 32 bits to 64 bits. As a result, a 4-KB PAE Page Table includes 512 entries instead of 1,024.

- A new level of Page Table called the Page Directory Pointer Table (PDPT) consisting of four 64-bit entries has been introduced.

- The cr3 control register contains a 27-bit Page Directory Pointer Table base address field. Because PDPTs are stored in the first 4 GB of RAM and aligned to a multiple of 32 bytes (2^5), 27 bits are sufficient to represent the base address of such tables.

- When mapping linear addresses to 4 KB pages (PS flag cleared in Page Directory entry), the 32 bits of a linear address are interpreted in the following way:

cr3
> Points to a PDPT

bits 31–30
> Point to 1 of 4 possible entries in PDPT

bits 29–21
> Point to 1 of 512 possible entries in Page Directory

bits 20–12
> Point to 1 of 512 possible entries in Page Table

bits 11–0
> Offset of 4-KB page

- When mapping linear addresses to 2-MB pages (PS flag set in Page Directory entry), the 32 bits of a linear address are interpreted in the following way:

cr3

Points to a PDPT

bits 31–30

Point to 1 of 4 possible entries in PDPT

bits 29–21

Point to 1 of 512 possible entries in Page Directory

bits 20–0

Offset of 2-MB page

To summarize, once cr3 is set, it is possible to address up to 4 GB of RAM. If we want to address more RAM, we'll have to put a new value in cr3 or change the content of the PDPT. However, the main problem with PAE is that linear addresses are still 32 bits long. This forces kernel programmers to reuse the same linear addresses to map different areas of RAM. We'll sketch how Linux initializes Page Tables when PAE is enabled in the later section, "Final kernel Page Table when RAM size is more than 4096 MB." Clearly, PAE does not enlarge the linear address space of a process, because it deals only with physical addresses. Furthermore, only the kernel can modify the page tables of the processes, thus a process running in User Mode cannot use a physical address space larger than 4 GB. On the other hand, PAE allows the kernel to exploit up to 64 GB of RAM, and thus to increase significantly the number of processes in the system.

Paging for 64-bit Architectures

As we have seen in the previous sections, two-level paging is commonly used by 32-bit microprocessors[*]. Two-level paging, however, is not suitable for computers that adopt a 64-bit architecture. Let's use a thought experiment to explain why:

Start by assuming a standard page size of 4 KB. Because 1 KB covers a range of 2^{10} addresses, 4 KB covers 2^{12} addresses, so the Offset field is 12 bits. This leaves up to 52 bits of the linear address to be distributed between the Table and the Directory fields. If we now decide to use only 48 of the 64 bits for addressing (this restriction leaves us with a comfortable 256 TB address space!), the remaining 48-12 = 36 bits will have to be split among Table and the Directory fields. If we now decide to reserve 18 bits for each of these two fields, both the Page Directory and the Page Tables of each process should include 2^{18} entries—that is, more than 256,000 entries.

---

[*] The third level of paging present in 80x86 processors with PAE enabled has been introduced only to lower from 1024 to 512 the number of entries in the Page Directory and Page Tables. This enlarges the Page Table entries from 32 bits to 64 bits so that they can store the 24 most significant bits of the physical address.

For that reason, all hardware paging systems for 64-bit processors make use of additional paging levels. The number of levels used depends on the type of processor. Table 2-4 summarizes the main characteristics of the hardware paging systems used by some 64-bit platforms supported by Linux. Please refer to the section "Hardware Dependency" in Chapter 1 for a short description of the hardware associated with the platform name.

Table 2-4. Paging levels in some 64-bit architectures

Platform name	Page size	Number of address bits used	Number of paging levels	Linear address splitting
alpha	8 KB [a]	43	3	10 + 10 + 10 + 13
ia64	4 KB [a]	39	3	9 + 9 + 9 + 12
ppc64	4 KB	41	3	10 + 10 + 9 + 12
sh64	4 KB	41	3	10 + 10 + 9 + 12
x86_64	4 KB	48	4	9 + 9 + 9 + 9 + 12

[a] This architecture supports different page sizes; we select a typical page size adopted by Linux.

As we will see in the section "Paging in Linux" later in this chapter, Linux succeeds in providing a common paging model that fits most of the supported hardware paging systems.

Hardware Cache

Today's microprocessors have clock rates of several gigahertz, while dynamic RAM (DRAM) chips have access times in the range of hundreds of clock cycles. This means that the CPU may be held back considerably while executing instructions that require fetching operands from RAM and/or storing results into RAM.

Hardware cache memories were introduced to reduce the speed mismatch between CPU and RAM. They are based on the well-known *locality principle*, which holds both for programs and data structures. This states that because of the cyclic structure of programs and the packing of related data into linear arrays, addresses close to the ones most recently used have a high probability of being used in the near future. It therefore makes sense to introduce a smaller and faster memory that contains the most recently used code and data. For this purpose, a new unit called the *line* was introduced into the 80×86 architecture. It consists of a few dozen contiguous bytes that are transferred in burst mode between the slow DRAM and the fast on-chip static RAM (SRAM) used to implement caches.

The cache is subdivided into subsets of lines. At one extreme, the cache can be *direct mapped*, in which case a line in main memory is always stored at the exact same location in the cache. At the other extreme, the cache is *fully associative*, meaning that any line in memory can be stored at any location in the cache. But most caches are to some degree *N-way set associative*, where any line of main memory can be stored in

any one of *N* lines of the cache. For instance, a line of memory can be stored in two different lines of a two-way set associative cache.

As shown in Figure 2-10, the cache unit is inserted between the paging unit and the main memory. It includes both a *hardware cache* memory and a *cache controller*. The cache memory stores the actual lines of memory. The cache controller stores an array of entries, one entry for each line of the cache memory. Each entry includes a *tag* and a few flags that describe the status of the cache line. The tag consists of some bits that allow the cache controller to recognize the memory location currently mapped by the line. The bits of the memory's physical address are usually split into three groups: the most significant ones correspond to the tag, the middle ones to the cache controller subset index, and the least significant ones to the offset within the line.
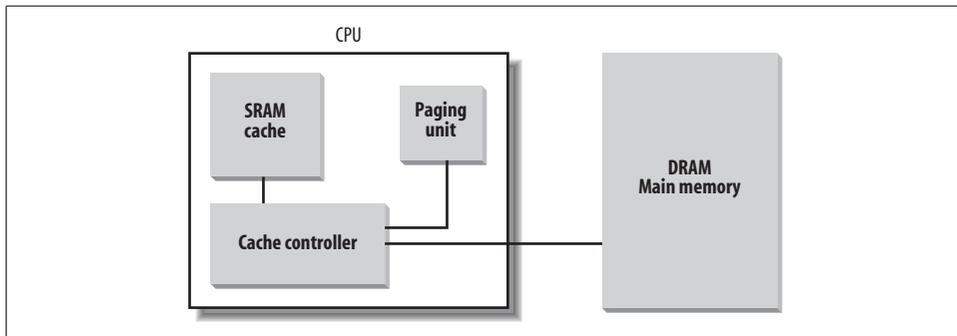
Figure 2-10. Processor hardware cache

When accessing a RAM memory cell, the CPU extracts the subset index from the physical address and compares the tags of all lines in the subset with the high-order bits of the physical address. If a line with the same tag as the high-order bits of the address is found, the CPU has a *cache hit*; otherwise, it has a *cache miss*.

When a cache hit occurs, the cache controller behaves differently, depending on the access type. For a read operation, the controller selects the data from the cache line and transfers it into a CPU register; the RAM is not accessed and the CPU saves time, which is why the cache system was invented. For a write operation, the controller may implement one of two basic strategies called *write-through* and *write-back*. In a write-through, the controller always writes into both RAM and the cache line, effectively switching off the cache for write operations. In a write-back, which offers more immediate efficiency, only the cache line is updated and the contents of the RAM are left unchanged. After a write-back, of course, the RAM must eventually be updated. The cache controller writes the cache line back into RAM only when the CPU executes an instruction requiring a flush of cache entries or when a FLUSH hardware signal occurs (usually after a cache miss).

When a cache miss occurs, the cache line is written to memory, if necessary, and the correct line is fetched from RAM into the cache entry.

Multiprocessor systems have a separate hardware cache for every processor, and therefore they need additional hardware circuitry to synchronize the cache contents. As shown in Figure 2-11, each CPU has its own local hardware cache. But now updating becomes more time consuming: whenever a CPU modifies its hardware cache, it must check whether the same data is contained in the other hardware cache; if so, it must notify the other CPU to update it with the proper value. This activity is often called *cache snooping*. Luckily, all this is done at the hardware level and is of no concern to the kernel.
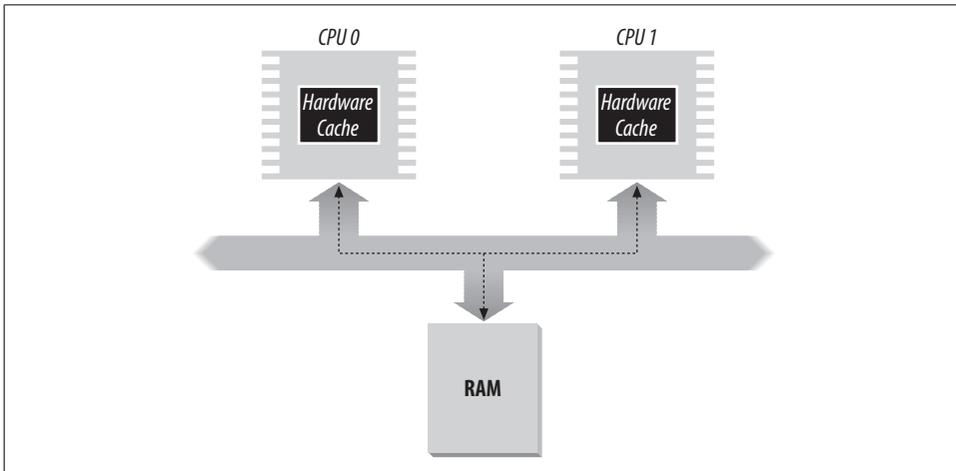
Figure 2-11. The caches in a dual processor

Cache technology is rapidly evolving. For example, the first Pentium models included a single on-chip cache called the *L1-cache*. More recent models also include other larger, slower on-chip caches called the *L2-cache*, *L3-cache*, etc. The consistency between the cache levels is implemented at the hardware level. Linux ignores these hardware details and assumes there is a single cache.

The CD flag of the cr0 processor register is used to enable or disable the cache circuitry. The NW flag, in the same register, specifies whether the write-through or the write-back strategy is used for the caches.

Another interesting feature of the Pentium cache is that it lets an operating system associate a different cache management policy with each page frame. For this purpose, each Page Directory and each Page Table entry includes two flags: PCD (Page Cache Disable), which specifies whether the cache must be enabled or disabled while accessing data included in the page frame; and PWT (Page Write-Through), which specifies whether the write-back or the write-through strategy must be applied while writing data into the page frame. Linux clears the PCD and PWT flags of all Page Directory and Page Table entries; as a result, caching is enabled for all page frames, and the write-back strategy is always adopted for writing.

Translation Lookaside Buffers (TLB)

Besides general-purpose hardware caches, 80×86 processors include another cache called *Translation Lookaside Buffers* (*TLB*) to speed up linear address translation. When a linear address is used for the first time, the corresponding physical address is computed through slow accesses to the Page Tables in RAM. The physical address is then stored in a TLB entry so that further references to the same linear address can be quickly translated.

In a multiprocessor system, each CPU has its own TLB, called the *local TLB* of the CPU. Contrary to the hardware cache, the corresponding entries of the TLB need not be synchronized, because processes running on the existing CPUs may associate the same linear address with different physical ones.

When the `cr3` control register of a CPU is modified, the hardware automatically invalidates all entries of the local TLB, because a new set of page tables is in use and the TLBs are pointing to old data.

Paging in Linux

Linux adopts a common paging model that fits both 32-bit and 64-bit architectures. As explained in the earlier section "Paging for 64-bit Architectures," two paging levels are sufficient for 32-bit architectures, while 64-bit architectures require a higher number of paging levels. Up to version 2.6.10, the Linux paging model consisted of three paging levels. Starting with version 2.6.11, a four-level paging model has been adopted.[*] The four types of page tables illustrated in Figure 2-12 are called:

- Page Global Directory
- Page Upper Directory
- Page Middle Directory
- Page Table

The Page Global Directory includes the addresses of several Page Upper Directories, which in turn include the addresses of several Page Middle Directories, which in turn include the addresses of several Page Tables. Each Page Table entry points to a page frame. Thus the linear address can be split into up to five parts. Figure 2-12 does not show the bit numbers, because the size of each part depends on the computer architecture.

For 32-bit architectures with no Physical Address Extension, two paging levels are sufficient. Linux essentially eliminates the Page Upper Directory and the Page Middle Directory fields by saying that they contain zero bits. However, the positions of

---

[*] This change has been made to fully support the linear address bit splitting used by the x86_64 platform (see Table 2-4).
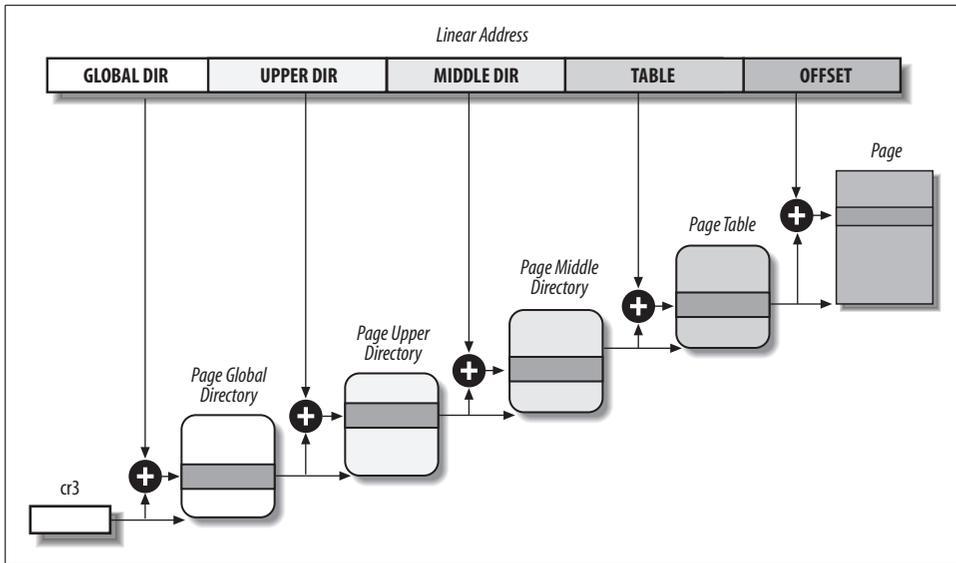
Figure 2-12. The Linux paging model

the Page Upper Directory and the Page Middle Directory in the sequence of pointers are kept so that the same code can work on 32-bit and 64-bit architectures. The kernel keeps a position for the Page Upper Directory and the Page Middle Directory by setting the number of entries in them to 1 and mapping these two entries into the proper entry of the Page Global Directory.

For 32-bit architectures with the Physical Address Extension enabled, three paging levels are used. The Linux's Page Global Directory corresponds to the 80×86's Page Directory Pointer Table, the Page Upper Directory is eliminated, the Page Middle Directory corresponds to the 80×86's Page Directory, and the Linux's Page Table corresponds to the 80×86's Page Table.

Finally, for 64-bit architectures three or four levels of paging are used depending on the linear address bit splitting performed by the hardware (see Table 2-4).

Linux's handling of processes relies heavily on paging. In fact, the automatic translation of linear addresses into physical ones makes the following design objectives feasible:

- Assign a different physical address space to each process, ensuring an efficient protection against addressing errors.
- Distinguish pages (groups of data) from page frames (physical addresses in main memory). This allows the same page to be stored in a page frame, then saved to disk and later reloaded in a different page frame. This is the basic ingredient of the virtual memory mechanism (see Chapter 17).

In the remaining part of this chapter, we will refer for the sake of concreteness to the paging circuitry used by the 80×86 processors.

As we will see in Chapter 9, each process has its own Page Global Directory and its own set of Page Tables. When a process switch occurs (see the section "Process Switch" in Chapter 3), Linux saves the `cr3` control register in the descriptor of the process previously in execution and then loads `cr3` with the value stored in the descriptor of the process to be executed next. Thus, when the new process resumes its execution on the CPU, the paging unit refers to the correct set of Page Tables.

Mapping linear to physical addresses now becomes a mechanical task, although it is still somewhat complex. The next few sections of this chapter are a rather tedious list of functions and macros that retrieve information the kernel needs to find addresses and manage the tables; most of the functions are one or two lines long. You may want to only skim these sections now, but it is useful to know the role of these functions and macros, because you'll see them often in discussions throughout this book.

The Linear Address Fields

The following macros simplify Page Table handling:

PAGE_SHIFT

> Specifies the length in bits of the Offset field; when applied to 80×86 processors, it yields the value 12. Because all the addresses in a page must fit in the Offset field, the size of a page on 80×86 systems is 2^{12} or the familiar 4,096 bytes; the PAGE_SHIFT of 12 can thus be considered the logarithm base 2 of the total page size. This macro is used by PAGE_SIZE to return the size of the page. Finally, the PAGE_MASK macro yields the value 0xfffff000 and is used to mask all the bits of the Offset field.

PMD_SHIFT

> The total length in bits of the Offset and Table fields of a linear address; in other words, the logarithm of the size of the area a Page Middle Directory entry can map. The PMD_SIZE macro computes the size of the area mapped by a single entry of the Page Middle Directory—that is, of a Page Table. The PMD_MASK macro is used to mask all the bits of the Offset and Table fields.
>
> When PAE is disabled, PMD_SHIFT yields the value 22 (12 from Offset plus 10 from Table), PMD_SIZE yields 2^{22} or 4 MB, and PMD_MASK yields 0xffc00000. Conversely, when PAE is enabled, PMD_SHIFT yields the value 21 (12 from Offset plus 9 from Table), PMD_SIZE yields 2^{21} or 2 MB, and PMD_MASK yields 0xffe00000.
>
> Large pages do not make use of the last level of page tables, thus LARGE_PAGE_SIZE, which yields the size of a large page, is equal to PMD_SIZE (2PMD_SHIFT) while LARGE_PAGE_MASK, which is used to mask all the bits of the Offset and Table fields in a large page address, is equal to PMD_MASK.

PUD_SHIFT

> Determines the logarithm of the size of the area a Page Upper Directory entry can map. The PUD_SIZE macro computes the size of the area mapped by a single entry of the Page Global Directory. The PUD_MASK macro is used to mask all the bits of the Offset, Table, Middle Air, and Upper Air fields.
>
> On the 80×86 processors, PUD_SHIFT is always equal to PMD_SHIFT and PUD_SIZE is equal to 4 MB or 2 MB.

PGDIR_SHIFT

> Determines the logarithm of the size of the area that a Page Global Directory entry can map. The PGDIR_SIZE macro computes the size of the area mapped by a single entry of the Page Global Directory. The PGDIR_MASK macro is used to mask all the bits of the Offset, Table, Middle Air, and Upper Air fields.
>
> When PAE is disabled, PGDIR_SHIFT yields the value 22 (the same value yielded by PMD_SHIFT and by PUD_SHIFT), PGDIR_SIZE yields 2^{22} or 4 MB, and PGDIR_MASK yields 0xffc00000. Conversely, when PAE is enabled, PGDIR_SHIFT yields the value 30 (12 from Offset plus 9 from Table plus 9 from Middle Air), PGDIR_SIZE yields 2^{30} or 1 GB, and PGDIR_MASK yields 0xc0000000.

PTRS_PER_PTE, PTRS_PER_PMD, PTRS_PER_PUD, *and* PTRS_PER_PGD

> Compute the number of entries in the Page Table, Page Middle Directory, Page Upper Directory, and Page Global Directory. They yield the values 1,024, 1, 1, and 1,024, respectively, when PAE is disabled; and the values 512, 512, 1, and 4, respectively, when PAE is enabled.

Page Table Handling

pte_t, pmd_t, pud_t, and pgd_t describe the format of, respectively, a Page Table, a Page Middle Directory, a Page Upper Directory, and a Page Global Directory entry. They are 64-bit data types when PAE is enabled and 32-bit data types otherwise. pgprot_t is another 64-bit (PAE enabled) or 32-bit (PAE disabled) data type that represents the protection flags associated with a single entry.

Five type-conversion macros—__pte, __pmd, __pud, __pgd, and __pgprot—cast an unsigned integer into the required type. Five other type-conversion macros—pte_val, pmd_val, pud_val, pgd_val, and pgprot_val—perform the reverse casting from one of the four previously mentioned specialized types into an unsigned integer.

The kernel also provides several macros and functions to read or modify page table entries:

- pte_none, pmd_none, pud_none, and pgd_none yield the value 1 if the corresponding entry has the value 0; otherwise, they yield the value 0.

- pte_clear, pmd_clear, pud_clear, and pgd_clear clear an entry of the corresponding page table, thus forbidding a process to use the linear addresses mapped by the page table entry. The ptep_get_and_clear() function clears a Page Table entry and returns the previous value.

- `set_pte`, `set_pmd`, `set_pud`, and `set_pgd` write a given value into a page table entry; `set_pte_atomic` is identical to `set_pte`, but when PAE is enabled it also ensures that the 64-bit value is written atomically.
- `pte_same(a,b)` returns 1 if two Page Table entries a and b refer to the same page and specify the same access privileges, 0 otherwise.
- `pmd_large(e)` returns 1 if the Page Middle Directory entry e refers to a large page (2 MB or 4 MB), 0 otherwise.

The `pmd_bad` macro is used by functions to check Page Middle Directory entries passed as input parameters. It yields the value 1 if the entry points to a bad Page Table—that is, if at least one of the following conditions applies:

- The page is not in main memory (`Present` flag cleared).
- The page allows only Read access (`Read/Write` flag cleared).
- Either `Accessed` or `Dirty` is cleared (Linux always forces these flags to be set for every existing Page Table).

The `pud_bad` and `pgd_bad` macros always yield 0. No `pte_bad` macro is defined, because it is legal for a Page Table entry to refer to a page that is not present in main memory, not writable, or not accessible at all.

The `pte_present` macro yields the value 1 if either the `Present` flag or the `Page Size` flag of a Page Table entry is equal to 1, the value 0 otherwise. Recall that the `Page Size` flag in Page Table entries has no meaning for the paging unit of the microprocessor; the kernel, however, marks `Present` equal to 0 and `Page Size` equal to 1 for the pages present in main memory but without read, write, or execute privileges. In this way, any access to such pages triggers a Page Fault exception because `Present` is cleared, and the kernel can detect that the fault is not due to a missing page by checking the value of `Page Size`.

The `pmd_present` macro yields the value 1 if the `Present` flag of the corresponding entry is equal to 1—that is, if the corresponding page or Page Table is loaded in main memory. The `pud_present` and `pgd_present` macros always yield the value 1.

The functions listed in Table 2-5 query the current value of any of the flags included in a Page Table entry; with the exception of `pte_file()`, these functions work properly only on Page Table entries for which `pte_present` returns 1.

Table 2-5. Page flag reading functions

Function name	Description
`pte_user()`	Reads the `User/Supervisor` flag
`pte_read()`	Reads the `User/Supervisor` flag (pages on the 80×86 processor cannot be protected against reading)
`pte_write()`	Reads the `Read/Write` flag
`pte_exec()`	Reads the `User/Supervisor` flag (pages on the 80 x 86 processor cannot be protected against code execution)

Table 2-5. Page flag reading functions (continued)

Function name	Description
pte_dirty( )	Reads the Dirty flag
pte_young( )	Reads the Accessed flag
pte_file( )	Reads the Dirty flag (when the Present flag is cleared and the Dirty flag is set, the page belongs to a non-linear disk file mapping; see Chapter 16)

Another group of functions listed in Table 2-6 sets the value of the flags in a Page Table entry.

Table 2-6. Page flag setting functions

Function name	Description
mk_pte_huge( )	Sets the Page Size and Present flags of a Page Table entry
pte_wrprotect( )	Clears the Read/Write flag
pte_rdprotect( )	Clears the User/Supervisor flag
pte_exprotect( )	Clears the User/Supervisor flag
pte_mkwrite( )	Sets the Read/Write flag
pte_mkread( )	Sets the User/Supervisor flag
pte_mkexec( )	Sets the User/Supervisor flag
pte_mkclean( )	Clears the Dirty flag
pte_mkdirty( )	Sets the Dirty flag
pte_mkold( )	Clears the Accessed flag (makes the page old)
pte_mkyoung( )	Sets the Accessed flag (makes the page young)
pte_modify(p,v)	Sets all access rights in a Page Table entry p to a specified value v
ptep_set_wrprotect( )	Like pte_wrprotect( ), but acts on a pointer to a Page Table entry
ptep_set_access_flags( )	If the Dirty flag is set, sets the page's access rights to a specified value and invokes flush_tlb_page( ) (see the section "Translation Lookaside Buffers (TLB)" later in this chapter)
ptep_mkdirty( )	Like pte_mkdirty( ) but acts on a pointer to a Page Table entry
ptep_test_and_clear_dirty( )	Like pte_mkclean( ) but acts on a pointer to a Page Table entry and returns the old value of the flag
ptep_test_and_clear_young( )	Like pte_mkold( ) but acts on a pointer to a Page Table entry and returns the old value of the flag

Now, let's discuss the macros listed in Table 2-7 that combine a page address and a group of protection flags into a page table entry or perform the reverse operation of extracting the page address from a page table entry. Notice that some of these mac-

ros refer to a page through the linear address of its "page descriptor" (see the section "Page Descriptors" in Chapter 8) rather than the linear address of the page itself.

Table 2-7. Macros acting on Page Table entries

Macro name	Description
pgd_index(addr)	Yields the index (relative position) of the entry in the Page Global Directory that maps the linear address addr.
pgd_offset(mm, addr)	Receives as parameters the address of a memory descriptor cw (see Chapter 9) and a linear address addr. The macro yields the linear address of the entry in a Page Global Directory that corresponds to the address addr; the Page Global Directory is found through a pointer within the memory descriptor.
pgd_offset_k(addr)	Yields the linear address of the entry in the master kernel Page Global Directory that corresponds to the address addr (see the later section "Kernel Page Tables").
pgd_page(pgd)	Yields the page descriptor address of the page frame containing the Page Upper Directory referred to by the Page Global Directory entry pgd. In a two- or three-level paging system, this macro is equivalent to pud_page() applied to the folded Page Upper Directory entry.
pud_offset(pgd, addr)	Receives as parameters a pointer pgd to a Page Global Directory entry and a linear address addr. The macro yields the linear address of the entry in a Page Upper Directory that corresponds to addr. In a two- or three-level paging system, this macro yields pgd, the address of a Page Global Directory entry.
pud_page(pud)	Yields the linear address of the Page Middle Directory referred to by the Page Upper Directory entry pud. In a two-level paging system, this macro is equivalent to pmd_page() applied to the folded Page Middle Directory entry.
pmd_index(addr)	Yields the index (relative position) of the entry in the Page Middle Directory that maps the linear address addr.
pmd_offset(pud, addr)	Receives as parameters a pointer pud to a Page Upper Directory entry and a linear address addr. The macro yields the address of the entry in a Page Middle Directory that corresponds to addr. In a two-level paging system, it yields pud, the address of a Page Global Directory entry.
pmd_page(pmd)	Yields the page descriptor address of the Page Table referred to by the Page Middle Directory entry pmd. In a two-level paging system, pmd is actually an entry of a Page Global Directory.
mk_pte(p,prot)	Receives as parameters the address of a page descriptor p and a group of access rights prot, and builds the corresponding Page Table entry.
pte_index(addr)	Yields the index (relative position) of the entry in the Page Table that maps the linear address addr.
pte_offset_kernel(dir, addr)	Yields the linear address of the Page Table that corresponds to the linear address addr mapped by the Page Middle Directory dir. Used only on the master kernel page tables (see the later section "Kernel Page Tables").

Table 2-7. Macros acting on Page Table entries (continued)

Macro name	Description
pte_offset_map(dir, addr)	Receives as parameters a pointer `dir` to a Page Middle Directory entry and a linear address `addr`; it yields the linear address of the entry in the Page Table that corresponds to the linear address `addr`. If the Page Table is kept in high memory, the kernel establishes a temporary kernel mapping (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8), to be released by means of `pte_unmap`. The macros `pte_offset_map_nested` and `pte_unmap_nested` are identical, but they use a different temporary kernel mapping.
pte_page(x)	Returns the page descriptor address of the page referenced by the Page Table entry `x`.
pte_to_pgoff(pte)	Extracts from the content `pte` of a Page Table entry the file offset corresponding to a page belonging to a non-linear file memory mapping (see the section "Non-Linear Memory Mappings" in Chapter 16).
pgoff_to_pte(offset)	Sets up the content of a Page Table entry for a page belonging to a non-linear file memory mapping.

The last group of functions of this long list was introduced to simplify the creation and deletion of page table entries.

When two-level paging is used, creating or deleting a Page Middle Directory entry is trivial. As we explained earlier in this section, the Page Middle Directory contains a single entry that points to the subordinate Page Table. Thus, the Page Middle Directory entry *is* the entry within the Page Global Directory, too. When dealing with Page Tables, however, creating an entry may be more complex, because the Page Table that is supposed to contain it might not exist. In such cases, it is necessary to allocate a new page frame, fill it with zeros, and add the entry.

If PAE is enabled, the kernel uses three-level paging. When the kernel creates a new Page Global Directory, it also allocates the four corresponding Page Middle Directories; these are freed only when the parent Page Global Directory is released.

When two or three-level paging is used, the Page Upper Directory entry is always mapped as a single entry within the Page Global Directory.

As usual, the description of the functions listed in Table 2-8 refers to the 80×86 architecture.

Table 2-8. Page allocation functions

Function name	Description
pgd_alloc(mm)	Allocates a new Page Global Directory; if PAE is enabled, it also allocates the three children Page Middle Directories that map the User Mode linear addresses. The argument mm (the address of a memory descriptor) is ignored on the 80×86 architecture.
pgd_free(pgd)	Releases the Page Global Directory at address pgd; if PAE is enabled, it also releases the three Page Middle Directories that map the User Mode linear addresses.

Table 2-8. Page allocation functions (continued)

Function name	Description
pud_alloc(mm, pgd, addr)	In a two- or three-level paging system, this function does nothing: it simply returns the linear address of the Page Global Directory entry pgd.
pud_free(x)	In a two- or three-level paging system, this macro does nothing.
pmd_alloc(mm, pud, addr)	Defined so generic three-level paging systems can allocate a new Page Middle Directory for the linear address addr. If PAE is not enabled, the function simply returns the input parameter pud—that is, the address of the entry in the Page Global Directory. If PAE is enabled, the function returns the linear address of the Page Middle Directory entry that maps the linear address addr. The argument cw is ignored.
pmd_free(x)	Does nothing, because Page Middle Directories are allocated and deallocated together with their parent Page Global Directory.
pte_alloc_map(mm, pmd, addr)	Receives as parameters the address of a Page Middle Directory entry pmd and a linear address addr, and returns the address of the Page Table entry corresponding to addr. If the Page Middle Directory entry is null, the function allocates a new Page Table by invoking pte_alloc_one( ). If a new Page Table is allocated, the entry corresponding to addr is initialized and the User/Supervisor flag is set. If the Page Table is kept in high memory, the kernel establishes a temporary kernel mapping (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8), to be released by pte_unmap.
pte_alloc_kernel(mm, pmd, addr)	If the Page Middle Directory entry pmd associated with the address addr is null, the function allocates a new Page Table. It then returns the linear address of the Page Table entry associated with addr. Used only for master kernel page tables (see the later section "Kernel Page Tables").
pte_free(pte)	Releases the Page Table associated with the pte page descriptor pointer.
pte_free_kernel(pte)	Equivalent to pte_free( ), but used for master kernel page tables.
clear_page_range(mmu, start,end)	Clears the contents of the page tables of a process from linear address start to end by iteratively releasing its Page Tables and clearing the Page Middle Directory entries.

Physical Memory Layout

During the initialization phase the kernel must build a *physical addresses map* that specifies which physical address ranges are usable by the kernel and which are unavailable (either because they map hardware devices' I/O shared memory or because the corresponding page frames contain BIOS data).

The kernel considers the following page frames as *reserved*:

- Those falling in the unavailable physical address ranges
- Those containing the kernel's code and initialized data structures

A page contained in a reserved page frame can never be dynamically assigned or swapped to disk.

As a general rule, the Linux kernel is installed in RAM starting from the physical address 0x00100000—i.e., from the second megabyte. The total number of page

frames required depends on how the kernel is configured. A typical configuration yields a kernel that can be loaded in less than 3 MB of RAM.

Why isn't the kernel loaded starting with the first available megabyte of RAM? Well, the PC architecture has several peculiarities that must be taken into account. For example:

- Page frame 0 is used by BIOS to store the system hardware configuration detected during the *Power-On Self-Test* (*POST*); the BIOS of many laptops, moreover, writes data on this page frame even after the system is initialized.

- Physical addresses ranging from 0x000a0000 to 0x000fffff are usually reserved to BIOS routines and to map the internal memory of ISA graphics cards. This area is the well-known hole from 640 KB to 1 MB in all IBM-compatible PCs: the physical addresses exist but they are reserved, and the corresponding page frames cannot be used by the operating system.

- Additional page frames within the first megabyte may be reserved by specific computer models. For example, the IBM ThinkPad maps the 0xa0 page frame into the 0x9f one.

In the early stage of the boot sequence (see Appendix A), the kernel queries the BIOS and learns the size of the physical memory. In recent computers, the kernel also invokes a BIOS procedure to build a list of physical address ranges and their corresponding memory types.

Later, the kernel executes the machine_specific_memory_setup() function, which builds the physical addresses map (see Table 2-9 for an example). Of course, the kernel builds this table on the basis of the BIOS list, if this is available; otherwise the kernel builds the table following the conservative default setup: all page frames with numbers from 0x9f (LOWMEMSIZE()) to 0x100 (HIGH_MEMORY) are marked as reserved.

Table 2-9. Example of BIOS-provided physical addresses map

Start	End	Type
0x00000000	0x0009ffff	Usable
0x000f0000	0x000fffff	Reserved
0x00100000	0x07feffff	Usable
0x07ff0000	0x07ff2fff	ACPI data
0x07ff3000	0x07ffffff	ACPI NVS
0xffff0000	0xffffffff	Reserved

A typical configuration for a computer having 128 MB of RAM is shown in Table 2-9. The physical address range from 0x07ff0000 to 0x07ff2fff stores information about the hardware devices of the system written by the BIOS in the POST phase; during the initialization phase, the kernel copies such information in a suitable kernel data structure, and then considers these page frames usable. Conversely, the physical address range of 0x07ff3000 to 0x07ffffff is mapped to ROM chips of

the hardware devices. The physical address range starting from 0xffff0000 is marked as reserved, because it is mapped by the hardware to the BIOS's ROM chip (see Appendix A). Notice that the BIOS may not provide information for some physical address ranges (in the table, the range is 0x000a0000 to 0x000effff). To be on the safe side, Linux assumes that such ranges are not usable.

The kernel might not see all physical memory reported by the BIOS: for instance, the kernel can address only 4 GB of RAM if it has not been compiled with PAE support, even if a larger amount of physical memory is actually available. The setup_memory( ) function is invoked right after machine_specific_memory_setup( ): it analyzes the table of physical memory regions and initializes a few variables that describe the kernel's physical memory layout. These variables are shown in Table 2-10.

Table 2-10. Variables describing the kernel's physical memory layout

Variable name	Description
num_physpages	Page frame number of the highest usable page frame
totalram_pages	Total number of usable page frames
min_low_pfn	Page frame number of the first usable page frame after the kernel image in RAM
max_pfn	Page frame number of the last usable page frame
max_low_pfn	Page frame number of the last page frame directly mapped by the kernel (low memory)
totalhigh_pages	Total number of page frames not directly mapped by the kernel (high memory)
highstart_pfn	Page frame number of the first page frame not directly mapped by the kernel
highend_pfn	Page frame number of the last page frame not directly mapped by the kernel

To avoid loading the kernel into groups of noncontiguous page frames, Linux prefers to skip the first megabyte of RAM. Clearly, page frames not reserved by the PC architecture will be used by Linux to store dynamically assigned pages.

Figure 2-13 shows how the first 3 MB of RAM are filled by Linux. We have assumed that the kernel requires less than 3 MB of RAM.

The symbol _text, which corresponds to physical address 0x00100000, denotes the address of the first byte of kernel code. The end of the kernel code is similarly identified by the symbol _etext. Kernel data is divided into two groups: *initialized* and *uninitialized*. The initialized data starts right after _etext and ends at _edata. The uninitialized data follows and ends up at _end.

The symbols appearing in the figure are not defined in Linux source code; they are produced while compiling the kernel.[*]

---

[*] You can find the linear address of these symbols in the file *System.map*, which is created right after the kernel is compiled.
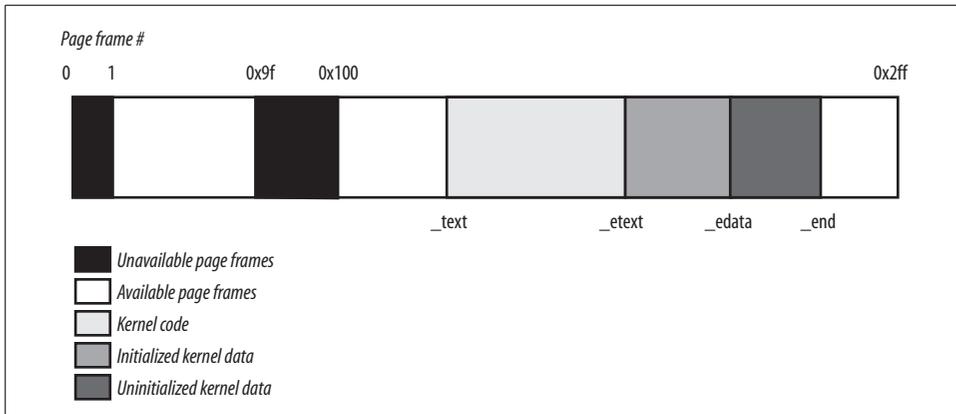
Figure 2-13. The first 768 page frames (3 MB) in Linux 2.6

Process Page Tables

The linear address space of a process is divided into two parts:

- Linear addresses from 0x00000000 to 0xbfffffff can be addressed when the process runs in either User or Kernel Mode.
- Linear addresses from 0xc0000000 to 0xffffffff can be addressed only when the process runs in Kernel Mode.

When a process runs in User Mode, it issues linear addresses smaller than 0xc0000000; when it runs in Kernel Mode, it is executing kernel code and the linear addresses issued are greater than or equal to 0xc0000000. In some cases, however, the kernel must access the User Mode linear address space to retrieve or store data.

The PAGE_OFFSET macro yields the value 0xc0000000; this is the offset in the linear address space of a process where the kernel lives. In this book, we often refer directly to the number 0xc0000000 instead.

The content of the first entries of the Page Global Directory that map linear addresses lower than 0xc0000000 (the first 768 entries with PAE disabled, or the first 3 entries with PAE enabled) depends on the specific process. Conversely, the remaining entries should be the same for all processes and equal to the corresponding entries of the master kernel Page Global Directory (see the following section).

Kernel Page Tables

The kernel maintains a set of page tables for its own use, rooted at a so-called *master kernel Page Global Directory*. After system initialization, this set of page tables is never directly used by any process or kernel thread; rather, the highest entries of the master kernel Page Global Directory are the reference model for the corresponding entries of the Page Global Directories of every regular process in the system.

We explain how the kernel ensures that changes to the master kernel Page Global Directory are propagated to the Page Global Directories that are actually used by processes in the section "Linear Addresses of Noncontiguous Memory Areas" in Chapter 8.

We now describe how the kernel initializes its own page tables. This is a two-phase activity. In fact, right after the kernel image is loaded into memory, the CPU is still running in real mode; thus, paging is not enabled.

In the first phase, the kernel creates a limited address space including the kernel's code and data segments, the initial Page Tables, and 128 KB for some dynamic data structures. This minimal address space is just large enough to install the kernel in RAM and to initialize its core data structures.

In the second phase, the kernel takes advantage of all of the existing RAM and sets up the page tables properly. Let us examine how this plan is executed.

Provisional kernel Page Tables

A provisional Page Global Directory is initialized statically during kernel compilation, while the provisional Page Tables are initialized by the startup_32() assembly language function defined in *arch/i386/kernel/head.S*. We won't bother mentioning the Page Upper Directories and Page Middle Directories anymore, because they are equated to Page Global Directory entries. PAE support is not enabled at this stage.

The provisional Page Global Directory is contained in the swapper_pg_dir variable. The provisional Page Tables are stored starting from pg0, right after the end of the kernel's uninitialized data segments (symbol _end in Figure 2-13). For the sake of simplicity, let's assume that the kernel's segments, the provisional Page Tables, and the 128 KB memory area fit in the first 8 MB of RAM. In order to map 8 MB of RAM, two Page Tables are required.

The objective of this first phase of paging is to allow these 8 MB of RAM to be easily addressed both in real mode and protected mode. Therefore, the kernel must create a mapping from both the linear addresses 0x00000000 through 0x007fffff and the linear addresses 0xc0000000 through 0xc07fffff into the physical addresses 0x00000000 through 0x007fffff. In other words, the kernel during its first phase of initialization can address the first 8 MB of RAM by either linear addresses identical to the physical ones or 8 MB worth of linear addresses, starting from 0xc0000000.

The Kernel creates the desired mapping by filling all the swapper_pg_dir entries with zeroes, except for entries 0, 1, 0x300 (decimal 768), and 0x301 (decimal 769); the latter two entries span all linear addresses between 0xc0000000 and 0xc07fffff. The 0, 1, 0x300, and 0x301 entries are initialized as follows:

- The address field of entries 0 and 0x300 is set to the physical address of pg0, while the address field of entries 1 and 0x301 is set to the physical address of the page frame following pg0.

- The `Present`, `Read/Write`, and `User/Supervisor` flags are set in all four entries.
- The `Accessed`, `Dirty`, `PCD`, `PWD`, and `Page Size` flags are cleared in all four entries.

The `startup_32()` assembly language function also enables the paging unit. This is achieved by loading the physical address of `swapper_pg_dir` into the `cr3` control register and by setting the `PG` flag of the `cr0` control register, as shown in the following equivalent code fragment:

```
movl $swapper_pg_dir-0xc0000000,%eax
movl %eax,%cr3          /* set the page table pointer.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0          /* ..and set paging (PG) bit */
```

Final kernel Page Table when RAM size is less than 896 MB

The final mapping provided by the kernel page tables must transform linear addresses starting from `0xc0000000` into physical addresses starting from 0.

The `__pa` macro is used to convert a linear address starting from `PAGE_OFFSET` to the corresponding physical address, while the `__va` macro does the reverse.

The master kernel Page Global Directory is still stored in `swapper_pg_dir`. It is initialized by the `paging_init()` function, which does the following:

1. Invokes `pagetable_init()` to set up the Page Table entries properly.
2. Writes the physical address of `swapper_pg_dir` in the `cr3` control register.
3. If the CPU supports PAE and if the kernel is compiled with PAE support, sets the `PAE` flag in the `cr4` control register.
4. Invokes `__flush_tlb_all()` to invalidate all TLB entries.

The actions performed by `pagetable_init()` depend on both the amount of RAM present and on the CPU model. Let's start with the simplest case. Our computer has less than 896 MB[*] of RAM, 32-bit physical addresses are sufficient to address all the available RAM, and there is no need to activate the PAE mechanism. (See the earlier section "The Physical Address Extension (PAE) Paging Mechanism.")

The `swapper_pg_dir` Page Global Directory is reinitialized by a cycle equivalent to the following:

```
pgd = swapper_pg_dir + pgd_index(PAGE_OFFSET); /* 768 */
phys_addr = 0x00000000;
while (phys_addr < (max_low_pfn * PAGE_SIZE)) {
    pmd = one_md_table_init(pgd); /* returns pgd itself */
    set_pmd(pmd, __pmd(phys_addr | pgprot_val(__pgprot(0x1e3))));
```

---

[*] The highest 128 MB of linear addresses are left available for several kinds of mappings (see sections "Fix-Mapped Linear Addresses" later in this chapter and "Linear Addresses of Noncontiguous Memory Areas" in Chapter 8). The kernel address space left for mapping the RAM is thus 1 GB – 128 MB = 896 MB.

```
        /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
                    Page Size, Global */
        phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x400000 */
        ++pgd;
    }
```

We assume that the CPU is a recent 80×86 microprocessor supporting 4 MB pages and "global" TLB entries. Notice that the User/Supervisor flags in all Page Global Directory entries referencing linear addresses above 0xc0000000 are cleared, thus denying processes in User Mode access to the kernel address space. Notice also that the Page Size flag is set so that the kernel can address the RAM by making use of large pages (see the section "Extended Paging" earlier in this chapter).

The identity mapping of the first megabytes of physical memory (8 MB in our example) built by the startup_32() function is required to complete the initialization phase of the kernel. When this mapping is no longer necessary, the kernel clears the corresponding page table entries by invoking the zap_low_mappings() function.

Actually, this description does not state the whole truth. As we'll see in the later section "Fix-Mapped Linear Addresses," the kernel also adjusts the entries of Page Tables corresponding to the "fix-mapped linear addresses."

Final kernel Page Table when RAM size is between 896 MB and 4096 MB

In this case, the RAM cannot be mapped entirely into the kernel linear address space. The best Linux can do during the initialization phase is to map a RAM window of size 896 MB into the kernel linear address space. If a program needs to address other parts of the existing RAM, some other linear address interval must be mapped to the required RAM. This implies changing the value of some page table entries. We'll discuss how this kind of dynamic remapping is done in Chapter 8.

To initialize the Page Global Directory, the kernel uses the same code as in the previous case.

Final kernel Page Table when RAM size is more than 4096 MB

Let's now consider kernel Page Table initialization for computers with more than 4 GB; more precisely, we deal with cases in which the following happens:

- The CPU model supports Physical Address Extension (PAE).
- The amount of RAM is larger than 4 GB.
- The kernel is compiled with PAE support.

Although PAE handles 36-bit physical addresses, linear addresses are still 32-bit addresses. As in the previous case, Linux maps a 896-MB RAM window into the kernel linear address space; the remaining RAM is left unmapped and handled by dynamic remapping, as described in Chapter 8. The main difference with the previ-

ous case is that a three-level paging model is used, so the Page Global Directory is initialized by a cycle equivalent to the following:

```
pgd_idx = pgd_index(PAGE_OFFSET); /* 3 */
for (i=0; i<pgd_idx; i++)
    set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001));
    /* 0x001 == Present */
pgd = swapper_pg_dir + pgd_idx;
phys_addr = 0x00000000;
for (; i<PTRS_PER_PGD; ++i, ++pgd) {
    pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
    set_pgd(pgd, __pgd(__pa(pmd) | 0x001)); /* 0x001 == Present */
    if (phys_addr < max_low_pfn * PAGE_SIZE)
        for (j=0; j < PTRS_PER_PMD /* 512 */
                && phys_addr < max_low_pfn*PAGE_SIZE; ++j) {
            set_pmd(pmd, __pmd(phys_addr |
                                pgprot_val(__pgprot(0x1e3))));
            /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
                    Page Size, Global */
            phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x200000 */
        }
}
swapper_pg_dir[0] = swapper_pg_dir[pgd_idx];
```

The kernel initializes the first three entries in the Page Global Directory corresponding to the user linear address space with the address of an empty page (empty_zero_page). The fourth entry is initialized with the address of a Page Middle Directory (pmd) allocated by invoking alloc_bootmem_low_pages(). The first 448 entries in the Page Middle Directory (there are 512 entries, but the last 64 are reserved for noncontiguous memory allocation; see the section "Noncontiguous Memory Area Management" in Chapter 8) are filled with the physical address of the first 896 MB of RAM.

Notice that all CPU models that support PAE also support large 2-MB pages and global pages. As in the previous cases, whenever possible, Linux uses large pages to reduce the number of Page Tables.

The fourth Page Global Directory entry is then copied into the first entry, so as to mirror the mapping of the low physical memory in the first 896 MB of the linear address space. This mapping is required in order to complete the initialization of SMP systems: when it is no longer necessary, the kernel clears the corresponding page table entries by invoking the zap_low_mappings() function, as in the previous cases.

Fix-Mapped Linear Addresses

We saw that the initial part of the fourth gigabyte of kernel linear addresses maps the physical memory of the system. However, at least 128 MB of linear addresses are always left available because the kernel uses them to implement noncontiguous memory allocation and fix-mapped linear addresses.

Noncontiguous memory allocation is just a special way to dynamically allocate and release pages of memory, and is described in the section "Linear Addresses of Non-contiguous Memory Areas" in Chapter 8. In this section, we focus on fix-mapped linear addresses.

Basically, a *fix-mapped linear address* is a constant linear address like `0xffffc000` whose corresponding physical address does not have to be the linear address minus `0xc000000`, but rather a physical address set in an arbitrary way. Thus, each fix-mapped linear address maps one page frame of the physical memory. As we'll see in later chapters, the kernel uses fix-mapped linear addresses instead of pointer variables that never change their value.

Fix-mapped linear addresses are conceptually similar to the linear addresses that map the first 896 MB of RAM. However, a fix-mapped linear address can map any physical address, while the mapping established by the linear addresses in the initial portion of the fourth gigabyte is linear (linear address X maps physical address X −`PAGE_OFFSET`).

With respect to variable pointers, fix-mapped linear addresses are more efficient. In fact, dereferencing a variable pointer requires one memory access more than dereferencing an immediate constant address. Moreover, checking the value of a variable pointer before dereferencing it is a good programming practice; conversely, the check is never required for a constant linear address.

Each fix-mapped linear address is represented by a small integer index defined in the `enum fixed_addresses` data structure:

```
enum fixed_addresses {
    FIX_HOLE,
    FIX_VSYSCALL,
    FIX_APIC_BASE,
    FIX_IO_APIC_BASE_0,
    [...]
    __end_of_fixed_addresses
};
```

Fix-mapped linear addresses are placed at the end of the fourth gigabyte of linear addresses. The `fix_to_virt()` function computes the constant linear address starting from the index:

```
inline unsigned long fix_to_virt(const unsigned int idx)
{
    if (idx >= __end_of_fixed_addresses)
        __this_fixmap_does_not_exist();
    return (0xfffff000UL - (idx << PAGE_SHIFT));
}
```

Let's assume that some kernel function invokes `fix_to_virt(FIX_IO_APIC_BASE_0)`. Because the function is declared as "inline," the C compiler does not generate a call to `fix_to_virt()`, but inserts its code in the calling function. Moreover, the check on

the index value is never performed at runtime. In fact, `FIX_IO_APIC_BASE_0` is a constant equal to 3, so the compiler can cut away the `if` statement because its condition is false at compile time. Conversely, if the condition is true or the argument of `fix_to_virt()` is not a constant, the compiler issues an error during the linking phase because the symbol `__this_fixmap_does_not_exist` is not defined anywhere. Eventually, the compiler computes `0xfffff000-(3<<PAGE_SHIFT)` and replaces the `fix_to_virt()` function call with the constant linear address `0xffffc000`.

To associate a physical address with a fix-mapped linear address, the kernel uses the `set_fixmap(idx,phys)` and `set_fixmap_nocache(idx,phys)` macros. Both of them initialize the Page Table entry corresponding to the `fix_to_virt(idx)` linear address with the physical address `phys`; however, the second function also sets the `PCD` flag of the Page Table entry, thus disabling the hardware cache when accessing the data in the page frame (see the section "Hardware Cache" earlier in this chapter). Conversely, `clear_fixmap(idx)` removes the linking between a fix-mapped linear address `idx` and the physical address.

Handling the Hardware Cache and the TLB

The last topic of memory addressing deals with how the kernel makes an optimal use of the hardware caches. Hardware caches and Translation Lookaside Buffers play a crucial role in boosting the performance of modern computer architectures. Several techniques are used by kernel developers to reduce the number of cache and TLB misses.

Handling the hardware cache

As mentioned earlier in this chapter, hardware caches are addressed by cache lines. The `L1_CACHE_BYTES` macro yields the size of a cache line in bytes. On Intel models earlier than the Pentium 4, the macro yields the value 32; on a Pentium 4, it yields the value 128.

To optimize the cache hit rate, the kernel considers the architecture in making the following decisions.

- The most frequently used fields of a data structure are placed at the low offset within the data structure, so they can be cached in the same line.
- When allocating a large set of data structures, the kernel tries to store each of them in memory in such a way that all cache lines are used uniformly.

Cache synchronization is performed automatically by the 80×86 microprocessors, thus the Linux kernel for this kind of processor does not perform any hardware

cache flushing. The kernel does provide, however, cache flushing interfaces for processors that do not synchronize caches.

Handling the TLB

Processors cannot synchronize their own TLB cache automatically because it is the kernel, and not the hardware, that decides when a mapping between a linear and a physical address is no longer valid.

Linux 2.6 offers several TLB flush methods that should be applied appropriately, depending on the type of page table change (see Table 2-11).

Table 2-11. Architecture-independent TLB-invalidating methods

Method name	Description	Typically used when
flush_tlb_all	Flushes all TLB entries (including those that refer to global pages, that is, pages whose Global flag is set)	Changing the kernel page table entries
flush_tlb_kernel_range	Flushes all TLB entries in a given range of linear addresses (including those that refer to global pages)	Changing a range of kernel page table entries
flush_tlb	Flushes all TLB entries of the non-global pages owned by the current process	Performing a process switch
flush_tlb_mm	Flushes all TLB entries of the non-global pages owned by a given process	Forking a new process
flush_tlb_range	Flushes the TLB entries corresponding to a linear address interval of a given process	Releasing a linear address interval of a process
flush_tlb_pgtables	Flushes the TLB entries of a given contiguous subset of page tables of a given process	Releasing some page tables of a process
flush_tlb_page	Flushes the TLB of a single Page Table entry of a given process	Processing a Page Fault

Despite the rich set of TLB methods offered by the generic Linux kernel, every microprocessor usually offers a far more restricted set of TLB-invalidating assembly language instructions. In this respect, one of the more flexible hardware platforms is Sun's UltraSPARC. In contrast, Intel microprocessors offers only two TLB-invalidating techniques:

- All Pentium models automatically flush the TLB entries relative to non-global pages when a value is loaded into the cr3 register.
- In Pentium Pro and later models, the invlpg assembly language instruction invalidates a single TLB entry mapping a given linear address.

Table 2-12 lists the Linux macros that exploit such hardware techniques; these macros are the basic ingredients to implement the architecture-independent methods listed in Table 2-11.

Table 2-12. TLB-invalidating macros for the Intel Pentium Pro and later processors

Macro name	Description	Used by
__flush_tlb()	Rewrites cr3 register back into itself	flush_tlb, flush_tlb_mm, flush_tlb_range
__flush_tlb_global()	Disables global pages by clearing the PGE flag of cr4, rewrites cr3 register back into itself, and sets again the PGE flag	flush_tlb_all, flush_tlb_kernel_range
__flush_tlb_single(addr)	Executes invlpg assembly language instruction with parameter addr	flush_tlb_page

Notice that the flush_tlb_pgtables method is missing from Table 2-12: in the 80×86 architecture nothing has to be done when a page table is unlinked from its parent table, thus the function implementing this method is empty.

The architecture-independent TLB-invalidating methods are extended quite simply to multiprocessor systems. The function running on a CPU sends an Interprocessor Interrupt (see "Interprocessor Interrupt Handling" in Chapter 4) to the other CPUs that forces them to execute the proper TLB-invalidating function.

As a general rule, any process switch implies changing the set of active page tables. Local TLB entries relative to the old page tables must be flushed; this is done automatically when the kernel writes the address of the new Page Global Directory into the cr3 control register. The kernel succeeds, however, in avoiding TLB flushes in the following cases:

- When performing a process switch between two regular processes that use the same set of page tables (see the section "The schedule( ) Function" in Chapter 7).

- When performing a process switch between a regular process and a kernel thread. In fact, we'll see in the section "Memory Descriptor of Kernel Threads" in Chapter 9, that kernel threads do not have their own set of page tables; rather, they use the set of page tables owned by the regular process that was scheduled last for execution on the CPU.

Besides process switches, there are other cases in which the kernel needs to flush some entries in a TLB. For instance, when the kernel assigns a page frame to a User Mode process and stores its physical address into a Page Table entry, it must flush any local TLB entry that refers to the corresponding linear address. On multiprocessor systems, the kernel also must flush the same TLB entry on the CPUs that are using the same set of page tables, if any.

To avoid useless TLB flushing in multiprocessor systems, the kernel uses a technique called *lazy TLB mode*. The basic idea is the following: if several CPUs are using the same page tables and a TLB entry must be flushed on all of them, then TLB flushing may, in some cases, be delayed on CPUs running kernel threads.

In fact, each kernel thread does not have its own set of page tables; rather, it makes use of the set of page tables belonging to a regular process. However, there is no need to invalidate a TLB entry that refers to a User Mode linear address, because no kernel thread accesses the User Mode address space.[*]

When some CPUs start running a kernel thread, the kernel sets it into lazy TLB mode. When requests are issued to clear some TLB entries, each CPU in lazy TLB mode does not flush the corresponding entries; however, the CPU remembers that its current process is running on a set of page tables whose TLB entries for the User Mode addresses are invalid. As soon as the CPU in lazy TLB mode switches to a regular process with a different set of page tables, the hardware automatically flushes the TLB entries, and the kernel sets the CPU back in non-lazy TLB mode. However, if a CPU in lazy TLB mode switches to a regular process that owns the same set of page tables used by the previously running kernel thread, then any deferred TLB invalidation must be effectively applied by the kernel. This "lazy" invalidation is effectively achieved by flushing all non-global TLB entries of the CPU.

Some extra data structures are needed to implement the lazy TLB mode. The `cpu_tlbstate` variable is a static array of `NR_CPUS` structures (the default value for this macro is 32; it denotes the maximum number of CPUs in the system) consisting of an `active_mm` field pointing to the memory descriptor of the current process (see Chapter 9) and a `state` flag that can assume only two values: `TLBSTATE_OK` (non-lazy TLB mode) or `TLBSTATE_LAZY` (lazy TLB mode). Furthermore, each memory descriptor includes a `cpu_vm_mask` field that stores the indices of the CPUs that should receive Interprocessor Interrupts related to TLB flushing. This field is meaningful only when the memory descriptor belongs to a process currently in execution.

When a CPU starts executing a kernel thread, the kernel sets the `state` field of its `cpu_tlbstate` element to `TLBSTATE_LAZY`; moreover, the `cpu_vm_mask` field of the active memory descriptor stores the indices of all CPUs in the system, including the one that is entering in lazy TLB mode. When another CPU wants to invalidate the TLB entries of all CPUs relative to a given set of page tables, it delivers an Interprocessor Interrupt to all CPUs whose indices are included in the `cpu_vm_mask` field of the corresponding memory descriptor.

When a CPU receives an Interprocessor Interrupt related to TLB flushing and verifies that it affects the set of page tables of its current process, it checks whether the

---

[*] By the way, the `flush_tlb_all` method does not use the lazy TLB mode mechanism; it is usually invoked whenever the kernel modifies a Page Table entry relative to the Kernel Mode address space.

state field of its `cpu_tlbstate` element is equal to `TLBSTATE_LAZY`. In this case, the kernel refuses to invalidate the TLB entries and removes the CPU index from the `cpu_vm_mask` field of the memory descriptor. This has two consequences:

- As long as the CPU remains in lazy TLB mode, it will not receive other Interprocessor Interrupts related to TLB flushing.
- If the CPU switches to another process that is using the same set of page tables as the kernel thread that is being replaced, the kernel invokes `__flush_tlb()` to invalidate all non-global TLB entries of the CPU.

Processes

The concept of a process is fundamental to any multiprogramming operating system. A process is usually defined as an instance of a program in execution; thus, if 16 users are running *vi* at once, there are 16 separate processes (although they can share the same executable code). Processes are often called *tasks* or *threads* in the Linux source code.

In this chapter, we discuss static properties of processes and then describe how process switching is performed by the kernel. The last two sections describe how processes can be created and destroyed. We also describe how Linux supports multithreaded applications—as mentioned in Chapter 1, it relies on so-called lightweight processes (LWP).

Processes, Lightweight Processes, and Threads

The term "process" is often used with several different meanings. In this book, we stick to the usual OS textbook definition: a *process* is an instance of a program in execution. You might think of it as the collection of data structures that fully describes how far the execution of the program has progressed.

Processes are like human beings: they are generated, they have a more or less significant life, they optionally generate one or more child processes, and eventually they die. A small difference is that sex is not really common among processes—each process has just one parent.

From the kernel's point of view, the purpose of a process is to act as an entity to which system resources (CPU time, memory, etc.) are allocated.

When a process is created, it is almost identical to its parent. It receives a (logical) copy of the parent's address space and executes the same code as the parent, beginning at the next instruction following the process creation system call. Although the parent and child may share the pages containing the program code (text), they have

separate copies of the data (stack and heap), so that changes by the child to a memory location are invisible to the parent (and vice versa).

While earlier Unix kernels employed this simple model, modern Unix systems do not. They support *multithreaded applications*—user programs having many relatively independent execution flows sharing a large portion of the application data structures. In such systems, a process is composed of several *user threads* (or simply *threads*), each of which represents an execution flow of the process. Nowadays, most multithreaded applications are written using standard sets of library functions called *pthread (POSIX thread) libraries*.

Older versions of the Linux kernel offered no support for multithreaded applications. From the kernel point of view, a multithreaded application was just a normal process. The multiple execution flows of a multithreaded application were created, handled, and scheduled entirely in User Mode, usually by means of a POSIX-compliant *pthread* library.

However, such an implementation of multithreaded applications is not very satisfactory. For instance, suppose a chess program uses two threads: one of them controls the graphical chessboard, waiting for the moves of the human player and showing the moves of the computer, while the other thread ponders the next move of the game. While the first thread waits for the human move, the second thread should run continuously, thus exploiting the thinking time of the human player. However, if the chess program is just a single process, the first thread cannot simply issue a blocking system call waiting for a user action; otherwise, the second thread is blocked as well. Instead, the first thread must employ sophisticated nonblocking techniques to ensure that the process remains runnable.

Linux uses *lightweight processes* to offer better support for multithreaded applications. Basically, two lightweight processes may share some resources, like the address space, the open files, and so on. Whenever one of them modifies a shared resource, the other immediately sees the change. Of course, the two processes must synchronize themselves when accessing the shared resource.

A straightforward way to implement multithreaded applications is to associate a lightweight process with each thread. In this way, the threads can access the same set of application data structures by simply sharing the same memory address space, the same set of open files, and so on; at the same time, each thread can be scheduled independently by the kernel so that one may sleep while another remains runnable. Examples of POSIX-compliant *pthread* libraries that use Linux's lightweight processes are *LinuxThreads*, *Native POSIX Thread Library* (*NPTL*), and IBM's *Next Generation Posix Threading Package* (*NGPT*).

POSIX-compliant multithreaded applications are best handled by kernels that support "thread groups." In Linux a *thread group* is basically a set of lightweight processes that implement a multithreaded application and act as a whole with regards to

some system calls such as getpid(), kill(), and _exit(). We are going to describe them at length later in this chapter.

Process Descriptor

To manage processes, the kernel must have a clear picture of what each process is doing. It must know, for instance, the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, and so on. This is the role of the *process descriptor*—a task_struct type structure whose fields contain all the information related to a single process.[*] As the repository of so much information, the process descriptor is rather complex. In addition to a large number of fields containing process attributes, the process descriptor contains several pointers to other data structures that, in turn, contain pointers to other structures. Figure 3-1 describes the Linux process descriptor schematically.

The six data structures on the right side of the figure refer to specific resources owned by the process. Most of these resources will be covered in future chapters. This chapter focuses on two types of fields that refer to the process state and to process parent/child relationships.

Process State

As its name implies, the state field of the process descriptor describes what is currently happening to the process. It consists of an array of flags, each of which describes a possible process state. In the current Linux version, these states are mutually exclusive, and hence exactly one flag of state always is set; the remaining flags are cleared. The following are the possible process states:

TASK_RUNNING
   The process is either executing on a CPU or waiting to be executed.

TASK_INTERRUPTIBLE
   The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process (put its state back to TASK_RUNNING).

TASK_UNINTERRUPTIBLE
   Like TASK_INTERRUPTIBLE, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used. It is valuable, however, under certain specific conditions in which a process must wait until a given event occurs without being interrupted. For instance, this state may be used

---

[*] The kernel also defines the task_t data type to be equivalent to struct task_struct.
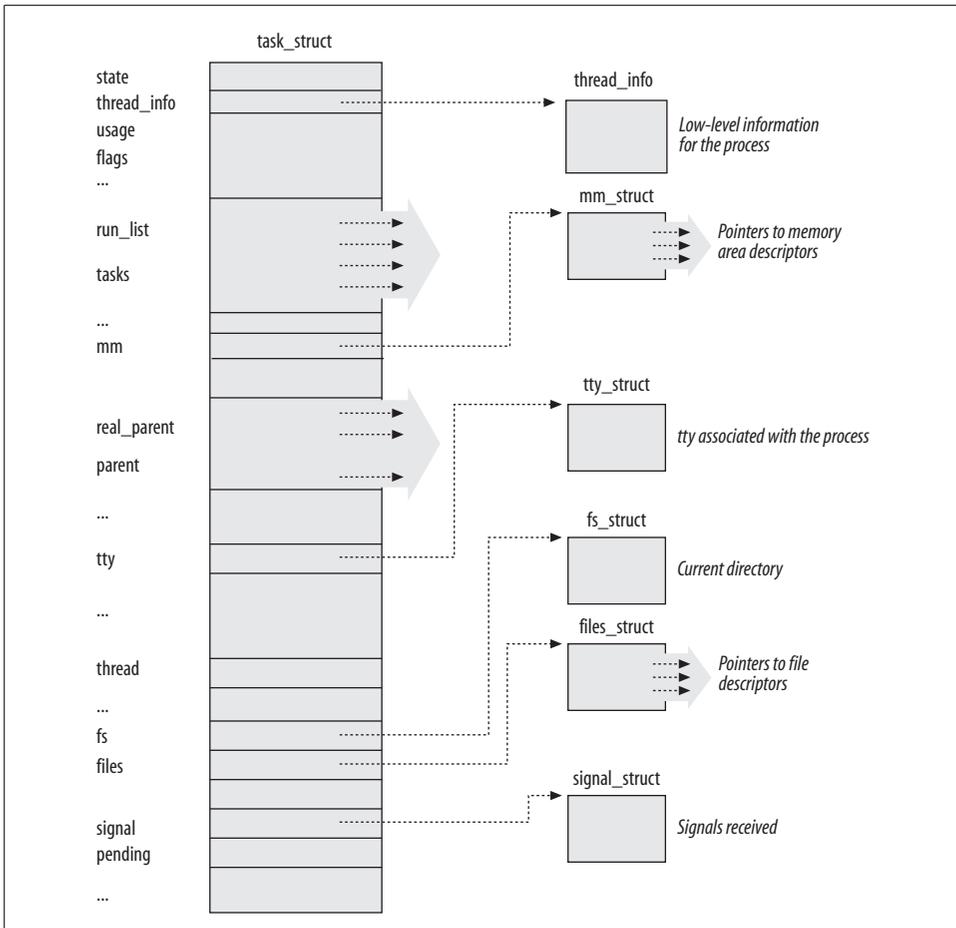
Figure 3-1. The Linux process descriptor

when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state.

TASK_STOPPED

Process execution has been stopped; the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal.

TASK_TRACED

Process execution has been stopped by a debugger. When a process is being monitored by another (such as when a debugger executes a ptrace( ) system call to monitor a test program), each signal may put the process in the TASK_TRACED state.

Two additional states of the process can be stored both in the state field and in the exit_state field of the process descriptor; as the field name suggests, a process reaches one of these two states only when its execution is terminated:

EXIT_ZOMBIE

Process execution is terminated, but the parent process has not yet issued a wait4() or waitpid() system call to return information about the dead process.* Before the wait()-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent might need it. (See the section "Process Removal" near the end of this chapter.)

EXIT_DEAD

The final state: the process is being removed by the system because the parent process has just issued a wait4() or waitpid() system call for it. Changing its state from EXIT_ZOMBIE to EXIT_DEAD avoids race conditions due to other threads of execution that execute wait()-like calls on the same process (see Chapter 5).

The value of the state field is usually set with a simple assignment. For instance:

```
p->state = TASK_RUNNING;
```

The kernel also uses the set_task_state and set_current_state macros: they set the state of a specified process and of the process currently executed, respectively. Moreover, these macros ensure that the assignment operation is not mixed with other instructions by the compiler or the CPU control unit. Mixing the instruction order may sometimes lead to catastrophic results (see Chapter 5).

Identifying a Process

As a general rule, each execution context that can be independently scheduled must have its own process descriptor; therefore, even lightweight processes, which share a large portion of their kernel data structures, have their own task_struct structures.

The strict one-to-one correspondence between the process and process descriptor makes the 32-bit address† of the task_struct structure a useful means for the kernel to identify processes. These addresses are referred to as *process descriptor pointers*. Most of the references to processes that the kernel makes are through process descriptor pointers.

On the other hand, Unix-like operating systems allow users to identify processes by means of a number called the *Process ID* (or *PID*), which is stored in the pid field of the process descriptor. PIDs are numbered sequentially: the PID of a newly created

---

* There are other wait()-like library functions, such as wait3() and wait(), but in Linux they are implemented by means of the wait4() and waitpid() system calls.

† As already noted in the section "Segmentation in Linux" in Chapter 2, although technically these 32 bits are only the offset component of a logical address, they coincide with the linear address.

process is normally the PID of the previously created process increased by one. Of course, there is an upper limit on the PID values; when the kernel reaches such limit, it must start recycling the lower, unused PIDs. By default, the maximum PID number is 32,767 (`PID_MAX_DEFAULT - 1`); the system administrator may reduce this limit by writing a smaller value into the */proc/sys/kernel/pid_max* file (*/proc* is the mount point of a special filesystem, see the section "Special Filesystems" in Chapter 12). In 64-bit architectures, the system administrator can enlarge the maximum PID number up to 4,194,303.

When recycling PID numbers, the kernel must manage a `pidmap_array` bitmap that denotes which are the PIDs currently assigned and which are the free ones. Because a page frame contains 32,768 bits, in 32-bit architectures the `pidmap_array` bitmap is stored in a single page. In 64-bit architectures, however, additional pages can be added to the bitmap when the kernel assigns a PID number too large for the current bitmap size. These pages are never released.

Linux associates a different PID with each process or lightweight process in the system. (As we shall see later in this chapter, there is a tiny exception on multiprocessor systems.) This approach allows the maximum flexibility, because every execution context in the system can be uniquely identified.

On the other hand, Unix programmers expect threads in the same group to have a common PID. For instance, it should be possible to a send a signal specifying a PID that affects all threads in the group. In fact, the POSIX 1003.1c standard states that all threads of a multithreaded application must have the same PID.

To comply with this standard, Linux makes use of thread groups. The identifier shared by the threads is the PID of the *thread group leader*, that is, the PID of the first lightweight process in the group; it is stored in the `tgid` field of the process descriptors. The `getpid()` system call returns the value of `tgid` relative to the current process instead of the value of `pid`, so all the threads of a multithreaded application share the same identifier. Most processes belong to a thread group consisting of a single member; as thread group leaders, they have the `tgid` field equal to the `pid` field, thus the `getpid()` system call works as usual for this kind of process.

Later, we'll show you how it is possible to derive a true process descriptor pointer efficiently from its respective PID. Efficiency is important because many system calls such as `kill()` use the PID to denote the affected process.

Process descriptors handling

Processes are dynamic entities whose lifetimes range from a few milliseconds to months. Thus, the kernel must be able to handle many processes at the same time, and process descriptors are stored in dynamic memory rather than in the memory area permanently assigned to the kernel. For each process, Linux packs two different

data structures in a single per-process memory area: a small data structure linked to the process descriptor, namely the `thread_info` structure, and the Kernel Mode process stack. The length of this memory area is usually 8,192 bytes (two page frames). For reasons of efficiency the kernel stores the 8-KB memory area in two consecutive page frames with the first page frame aligned to a multiple of 2^{13}; this may turn out to be a problem when little dynamic memory is available, because the free memory may become highly fragmented (see the section "The Buddy System Algorithm" in Chapter 8). Therefore, in the 80×86 architecture the kernel can be configured at compilation time so that the memory area including stack and `thread_info` structure spans a single page frame (4,096 bytes).

In the section "Segmentation in Linux" in Chapter 2, we learned that a process in Kernel Mode accesses a stack contained in the kernel data segment, which is different from the stack used by the process in User Mode. Because kernel control paths make little use of the stack, only a few thousand bytes of kernel stack are required. Therefore, 8 KB is ample space for the stack and the `thread_info` structure. However, when stack and `thread_info` structure are contained in a single page frame, the kernel uses a few additional stacks to avoid the overflows caused by deeply nested interrupts and exceptions (see Chapter 4).

Figure 3-2 shows how the two data structures are stored in the 2-page (8 KB) memory area. The `thread_info` structure resides at the beginning of the memory area, and the stack grows downward from the end. The figure also shows that the `thread_info` structure and the `task_struct` structure are mutually linked by means of the fields `task` and `thread_info`, respectively.
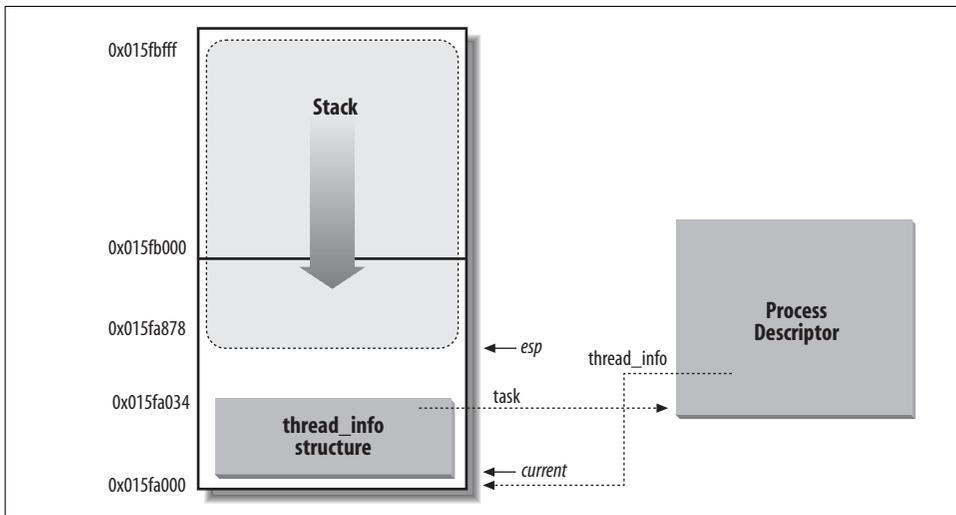
Figure 3-2. Storing the thread_info structure and the process kernel stack in two page frames

The esp register is the CPU stack pointer, which is used to address the stack's top location. On 80×86 systems, the stack starts at the end and grows toward the beginning of the memory area. Right after switching from User Mode to Kernel Mode, the kernel stack of a process is always empty, and therefore the esp register points to the byte immediately following the stack.

The value of the esp is decreased as soon as data is written into the stack. Because the thread_info structure is 52 bytes long, the kernel stack can expand up to 8,140 bytes.

The C language allows the thread_info structure and the kernel stack of a process to be conveniently represented by means of the following union construct:

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[2048]; /* 1024 for 4KB stacks */
};
```

The thread_info structure shown in Figure 3-2 is stored starting at address 0x015fa000, and the stack is stored starting at address 0x015fc000. The value of the esp register points to the current top of the stack at 0x015fa878.

The kernel uses the alloc_thread_info and free_thread_info macros to allocate and release the memory area storing a thread_info structure and a kernel stack.

Identifying the current process

The close association between the thread_info structure and the Kernel Mode stack just described offers a key benefit in terms of efficiency: the kernel can easily obtain the address of the thread_info structure of the process currently running on a CPU from the value of the esp register. In fact, if the thread_union structure is 8 KB (2^{13} bytes) long, the kernel masks out the 13 least significant bits of esp to obtain the base address of the thread_info structure; on the other hand, if the thread_union structure is 4 KB long, the kernel masks out the 12 least significant bits of esp. This is done by the current_thread_info() function, which produces assembly language instructions like the following:

```
movl $0xffffe000,%ecx /* or 0xfffff000 for 4KB stacks */
andl %esp,%ecx
movl %ecx,p
```

After executing these three instructions, p contains the thread_info structure pointer of the process running on the CPU that executes the instruction.

Most often the kernel needs the address of the process descriptor rather than the address of the thread_info structure. To get the process descriptor pointer of the process currently running on a CPU, the kernel makes use of the current macro,

which is essentially equivalent to `current_thread_info()->task` and produces assembly language instructions like the following:

```
movl $0xffffe000,%ecx /* or 0xfffff000 for 4KB stacks */
andl %esp,%ecx
movl (%ecx),p
```

Because the `task` field is at offset 0 in the `thread_info` structure, after executing these three instructions `p` contains the process descriptor pointer of the process running on the CPU.

The `current` macro often appears in kernel code as a prefix to fields of the process descriptor. For example, `current->pid` returns the process ID of the process currently running on the CPU.

Another advantage of storing the process descriptor with the stack emerges on multiprocessor systems: the correct current process for each hardware processor can be derived just by checking the stack, as shown previously. Earlier versions of Linux did not store the kernel stack and the process descriptor together. Instead, they were forced to introduce a global static variable called `current` to identify the process descriptor of the running process. On multiprocessor systems, it was necessary to define `current` as an array—one element for each available CPU.

Doubly linked lists

Before moving on and describing how the kernel keeps track of the various processes in the system, we would like to emphasize the role of special data structures that implement doubly linked lists.

For each list, a set of primitive operations must be implemented: initializing the list, inserting and deleting an element, scanning the list, and so on. It would be both a waste of programmers' efforts and a waste of memory to replicate the primitive operations for each different list.

Therefore, the Linux kernel defines the `list_head` data structure, whose only fields `next` and `prev` represent the forward and back pointers of a generic doubly linked list element, respectively. It is important to note, however, that the pointers in a `list_head` field store the addresses of other `list_head` fields rather than the addresses of the whole data structures in which the `list_head` structure is included; see Figure 3-3 (a).

A new list is created by using the `LIST_HEAD(list_name)` macro. It declares a new variable named `list_name` of type `list_head`, which is a dummy first element that acts as a placeholder for the head of the new list, and initializes the `prev` and `next` fields of the `list_head` data structure so as to point to the `list_name` variable itself; see Figure 3-3 (b).
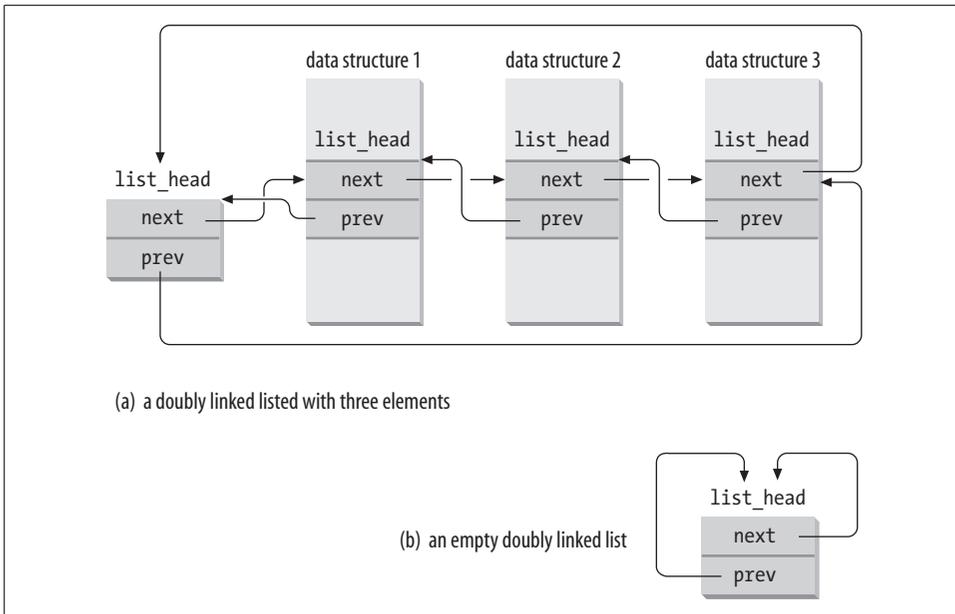
Figure 3-3. Doubly linked lists built with list_head data structures

Several functions and macros implement the primitives, including those shown in Table 3-1.

Table 3-1. List handling functions and macros

Name	Description
list_add(n,p)	Inserts an element pointed to by n right after the specified element pointed to by p. (To insert n at the beginning of the list, set p to the address of the list head.)
list_add_tail(n,p)	Inserts an element pointed to by n right before the specified element pointed to by p. (To insert n at the end of the list, set p to the address of the list head.)
list_del(p)	Deletes an element pointed to by p. (There is no need to specify the head of the list.)
list_empty(p)	Checks if the list specified by the address p of its head is empty.
list_entry(p,t,m)	Returns the address of the data structure of type t in which the list_head field that has the name m and the address p is included.
list_for_each(p,h)	Scans the elements of the list specified by the address h of the head; in each iteration, a pointer to the list_head structure of the list element is returned in p.
list_for_each_entry(p,h,m)	Similar to list_for_each, but returns the address of the data structure embedding the list_head structure rather than the address of the list_head structure itself.

The Linux kernel 2.6 sports another kind of doubly linked list, which mainly differs from a list_head list because it is not circular; it is mainly used for hash tables, where space is important, and finding the the last element in constant time is not. The list head is stored in an hlist_head data structure, which is simply a pointer to the first element in the list (NULL if the list is empty). Each element is represented by an hlist_node data structure, which includes a pointer next to the next element, and a pointer pprev to the next field of the previous element. Because the list is not circular, the pprev field of the first element and the next field of the last element are set to NULL. The list can be handled by means of several helper functions and macros similar to those listed in Table 3-1: hlist_add_head(), hlist_del(), hlist_empty(), hlist_entry, hlist_for_each_entry, and so on.

The process list

The first example of a doubly linked list we will examine is the *process list*, a list that links together all existing process descriptors. Each task_struct structure includes a tasks field of type list_head whose prev and next fields point, respectively, to the previous and to the next task_struct element.

The head of the process list is the init_task task_struct descriptor; it is the process descriptor of the so-called *process 0* or *swapper* (see the section "Kernel Threads" later in this chapter). The tasks->prev field of init_task points to the tasks field of the process descriptor inserted last in the list.

The SET_LINKS and REMOVE_LINKS macros are used to insert and to remove a process descriptor in the process list, respectively. These macros also take care of the parent-hood relationship of the process (see the section "How Processes Are Organized" later in this chapter).

Another useful macro, called for_each_process, scans the whole process list. It is defined as:

```
#define for_each_process(p) \
    for (p=&init_task; (p=list_entry((p)->tasks.next, \
                                struct task_struct, tasks) \
                            ) != &init_task; )
```

The macro is the loop control statement after which the kernel programmer supplies the loop. Notice how the init_task process descriptor just plays the role of list header. The macro starts by moving past init_task to the next task and continues until it reaches init_task again (thanks to the circularity of the list). At each iteration, the variable passed as the argument of the macro contains the address of the currently scanned process descriptor, as returned by the list_entry macro.

The lists of TASK_RUNNING processes

When looking for a new process to run on a CPU, the kernel has to consider only the runnable processes (that is, the processes in the TASK_RUNNING state).

Earlier Linux versions put all runnable processes in the same list called *runqueue*. Because it would be too costly to maintain the list ordered according to process priorities, the earlier schedulers were compelled to scan the whole list in order to select the "best" runnable process.

Linux 2.6 implements the runqueue differently. The aim is to allow the scheduler to select the best runnable process in constant time, independently of the number of runnable processes. We'll defer to Chapter 7 a detailed description of this new kind of runqueue, and we'll provide here only some basic information.

The trick used to achieve the scheduler speedup consists of splitting the runqueue in many lists of runnable processes, one list per process priority. Each task_struct descriptor includes a run_list field of type list_head. If the process priority is equal to k (a value ranging between 0 and 139), the run_list field links the process descriptor into the list of runnable processes having priority k. Furthermore, on a multiprocessor system, each CPU has its own runqueue, that is, its own set of lists of processes. This is a classic example of making a data structures more complex to improve performance: to make scheduler operations more efficient, the runqueue list has been split into 140 different lists!

As we'll see, the kernel must preserve a lot of data for every runqueue in the system; however, the main data structures of a runqueue are the lists of process descriptors belonging to the runqueue; all these lists are implemented by a single prio_array_t data structure, whose fields are shown in Table 3-2.

Table 3-2. The fields of the prio_array_t data structure

Type	Field	Description
int	nr_active	The number of process descriptors linked into the lists
unsigned long [5]	bitmap	A priority bitmap: each flag is set if and only if the corresponding priority list is not empty
struct list_head [140]	queue	The 140 heads of the priority lists

The enqueue_task(p,array) function inserts a process descriptor into a runqueue list; its code is essentially equivalent to:

```
list_add_tail(&p->run_list, &array->queue[p->prio]);
__set_bit(p->prio, array->bitmap);
array->nr_active++;
p->array = array;
```

The prio field of the process descriptor stores the dynamic priority of the process, while the array field is a pointer to the prio_array_t data structure of its current runqueue. Similarly, the dequeue_task(p,array) function removes a process descriptor from a runqueue list.

Relationships Among Processes

Processes created by a program have a parent/child relationship. When a process creates multiple children, these children have sibling relationships. Several fields must be introduced in a process descriptor to represent these relationships; they are listed in Table 3-3 with respect to a given process P. Processes 0 and 1 are created by the kernel; as we'll see later in the chapter, process 1 (*init*) is the ancestor of all other processes.

Table 3-3. Fields of a process descriptor used to express parenthood relationships

Field name	Description
`real_parent`	Points to the process descriptor of the process that created P or to the descriptor of process 1 (*init*) if the parent process no longer exists. (Therefore, when a user starts a background process and exits the shell, the background process becomes the child of *init*.)
`parent`	Points to the current parent of P (this is the process that must be signaled when the child process terminates); its value usually coincides with that of `real_parent`. It may occasionally differ, such as when another process issues a `ptrace( )` system call requesting that it be allowed to monitor P (see the section "Execution Tracing" in Chapter 20).
`children`	The head of the list containing all children created by P.
`sibling`	The pointers to the next and previous elements in the list of the sibling processes, those that have the same parent as P.

Figure 3-4 illustrates the parent and sibling relationships of a group of processes. Process P0 successively created P1, P2, and P3. Process P3, in turn, created process P4.

Furthermore, there exist other relationships among processes: a process can be a leader of a process group or of a login session (see "Process Management" in Chapter 1), it can be a leader of a thread group (see "Identifying a Process" earlier in this chapter), and it can also trace the execution of other processes (see the section "Execution Tracing" in Chapter 20). Table 3-4 lists the fields of the process descriptor that establish these relationships between a process P and the other processes.

Table 3-4. The fields of the process descriptor that establish non-parenthood relationships

Field name	Description
`group_leader`	Process descriptor pointer of the group leader of P
`signal->pgrp`	PID of the group leader of P
`tgid`	PID of the thread group leader of P
`signal->session`	PID of the login session leader of P
`ptrace_children`	The head of a list containing all children of P being traced by a debugger
`ptrace_list`	The pointers to the next and previous elements in the real parent's list of traced processes (used when P is being traced)
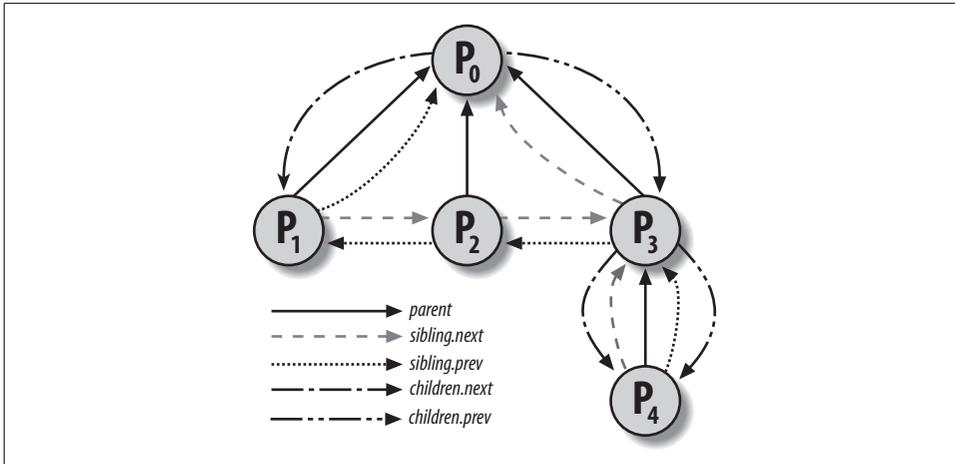
Figure 3-4. Parenthood relationships among five processes

The pidhash table and chained lists

In several circumstances, the kernel must be able to derive the process descriptor pointer corresponding to a PID. This occurs, for instance, in servicing the kill( ) system call. When process P1 wishes to send a signal to another process, P2, it invokes the kill( ) system call specifying the PID of P2 as the parameter. The kernel derives the process descriptor pointer from the PID and then extracts the pointer to the data structure that records the pending signals from P2's process descriptor.

Scanning the process list sequentially and checking the pid fields of the process descriptors is feasible but rather inefficient. To speed up the search, four hash tables have been introduced. Why multiple hash tables? Simply because the process descriptor includes fields that represent different types of PID (see Table 3-5), and each type of PID requires its own hash table.

Table 3-5. The four hash tables and corresponding fields in the process descriptor

Hash table type	Field name	Description
PIDTYPE_PID	pid	PID of the process
PIDTYPE_TGID	tgid	PID of thread group leader process
PIDTYPE_PGID	pgrp	PID of the group leader process
PIDTYPE_SID	session	PID of the session leader process

The four hash tables are dynamically allocated during the kernel initialization phase, and their addresses are stored in the pid_hash array. The size of a single hash table depends on the amount of available RAM; for example, for systems having 512 MB of RAM, each hash table is stored in four page frames and includes 2,048 entries.

The PID is transformed into a table index using the `pid_hashfn` macro, which expands to:

```
#define pid_hashfn(x) hash_long((unsigned long) x, pidhash_shift)
```

The `pidhash_shift` variable stores the length in bits of a table index (11, in our example). The `hash_long()` function is used by many hash functions; on a 32-bit architecture it is essentially equivalent to:

```
unsigned long hash_long(unsigned long val, unsigned int bits)
{
    unsigned long hash = val * 0x9e370001UL;
    return hash >> (32 - bits);
}
```

Because in our example `pidhash_shift` is equal to 11, `pid_hashfn` yields values ranging between 0 and $2^{11} - 1 = 2047$.

The Magic Constant

You might wonder where the 0x9e370001 constant (= 2,654,404,609) comes from. This hash function is based on a multiplication of the index by a suitable large number, so that the result overflows and the value remaining in the 32-bit variable can be considered as the result of a modulus operation. Knuth suggested that good results are obtained when the large multiplier is a prime approximately in golden ratio to 2^{32} (32 bit being the size of the 80×86's registers). Now, 2,654,404,609 is a prime near to $2^{32} \times (\sqrt{5} - 1)/2$ that can also be easily multiplied by additions and bit shifts, because it is equal to $2^{31} + 2^{29} - 2^{25} + 2^{22} - 2^{19} - 2^{16} + 1$.

As every basic computer science course explains, a hash function does not always ensure a one-to-one correspondence between PIDs and table indexes. Two different PIDs that hash into the same table index are said to be *colliding*.

Linux uses *chaining* to handle colliding PIDs; each table entry is the head of a doubly linked list of colliding process descriptors. Figure 3-5 illustrates a PID hash table with two lists. The processes having PIDs 2,890 and 29,384 hash into the 200th element of the table, while the process having PID 29,385 hashes into the 1,466th element of the table.

Hashing with chaining is preferable to a linear transformation from PIDs to table indexes because at any given instance, the number of processes in the system is usually far below 32,768 (the maximum number of allowed PIDs). It would be a waste of storage to define a table consisting of 32,768 entries, if, at any given instance, most such entries are unused.

The data structures used in the PID hash tables are quite sophisticated, because they must keep track of the relationships between the processes. As an example, suppose
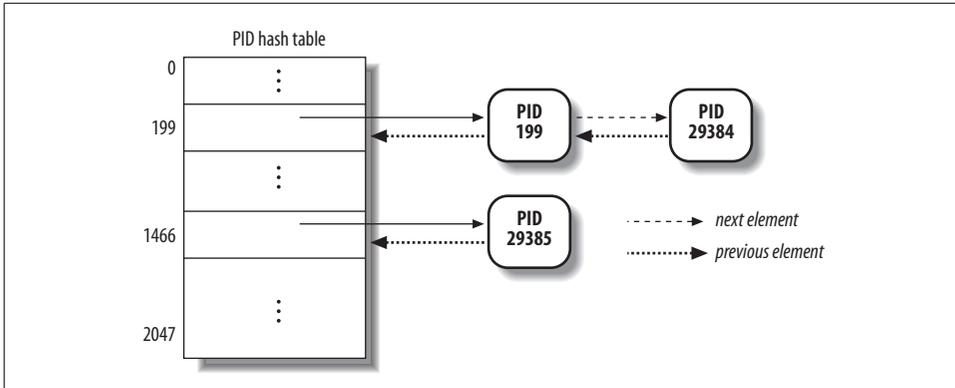
Figure 3-5. A simple PID hash table and chained lists

that the kernel must retrieve all processes belonging to a given thread group, that is, all processes whose `tgid` field is equal to a given number. Looking in the hash table for the given thread group number returns just one process descriptor, that is, the descriptor of the thread group leader. To quickly retrieve the other processes in the group, the kernel must maintain a list of processes for each thread group. The same situation arises when looking for the processes belonging to a given login session or belonging to a given process group.

The PID hash tables' data structures solve all these problems, because they allow the definition of a list of processes for any PID number included in a hash table. The core data structure is an array of four `pid` structures embedded in the `pids` field of the process descriptor; the fields of the `pid` structure are shown in Table 3-6.

Table 3-6. The fields of the pid data structures

Type	Name	Description
int	nr	The PID number
struct hlist_node	pid_chain	The links to the next and previous elements in the hash chain list
struct list_head	pid_list	The head of the per-PID list

Figure 3-6 shows an example based on the `PIDTYPE_TGID` hash table. The second entry of the `pid_hash` array stores the address of the hash table, that is, the array of `hlist_head` structures representing the heads of the chain lists. In the chain list rooted at the 71st entry of the hash table, there are two process descriptors corresponding to the PID numbers 246 and 4,351 (double-arrow lines represent a couple of forward and backward pointers). The PID numbers are stored in the `nr` field of the `pid` structure embedded in the process descriptor (by the way, because the thread group number coincides with the PID of its leader, these numbers also are stored in the `pid` field of the process descriptors). Let us consider the per-PID list of the thread group 4,351: the head of the list is stored in the `pid_list` field of the process descriptor included in
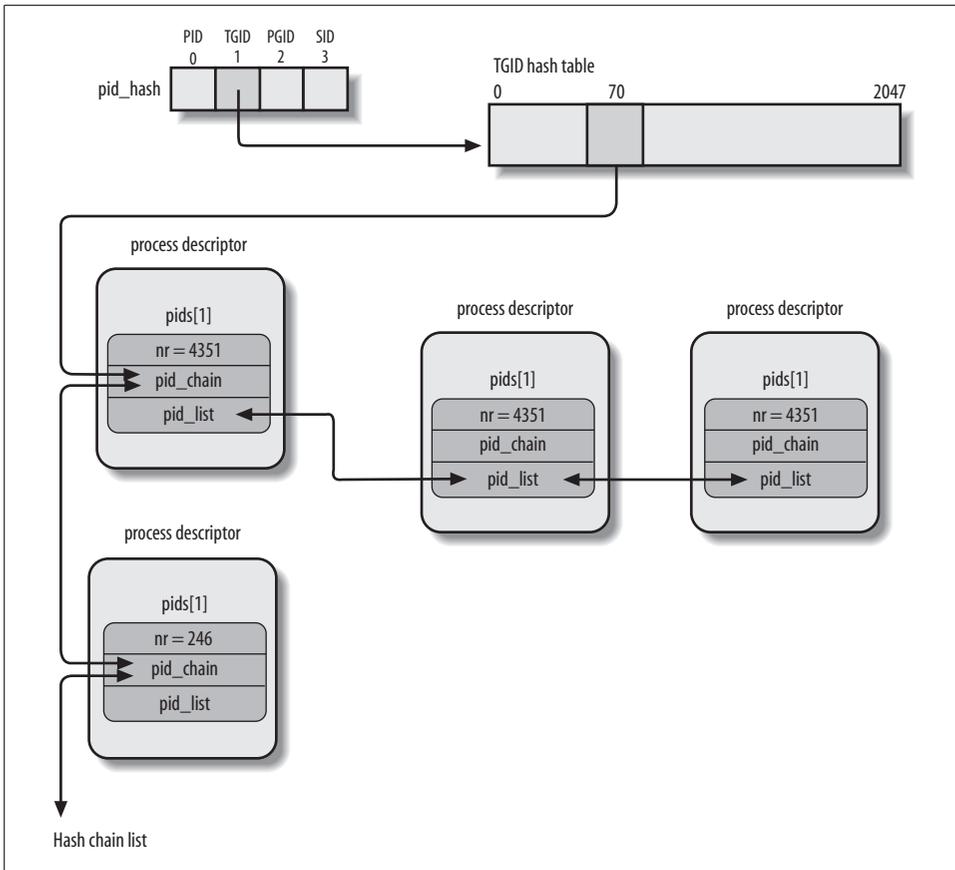
Figure 3-6. The PID hash tables

the hash table, while the links to the next and previous elements of the per-PID list also are stored in the pid_list field of each list element.

The following functions and macros are used to handle the PID hash tables:

do_each_task_pid(nr, type, task)
while_each_task_pid(nr, type, task)
> Mark begin and end of a do-while loop that iterates over the per-PID list associated with the PID number nr of type type; in any iteration, task points to the process descriptor of the currently scanned element.

find_task_by_pid_type(type, nr)
> Looks for the process having PID nr in the hash table of type type. The function returns a process descriptor pointer if a match is found, otherwise it returns NULL.

find_task_by_pid(nr)
> Same as find_task_by_pid_type(PIDTYPE_PID, nr).

`attach_pid(task, type, nr)`

> Inserts the process descriptor pointed to by `task` in the PID hash table of type `type` according to the PID number `nr`; if a process descriptor having PID `nr` is already in the hash table, the function simply inserts `task` in the per-PID list of the already present process.

`detach_pid(task, type)`

> Removes the process descriptor pointed to by `task` from the per-PID list of type `type` to which the descriptor belongs. If the per-PID list does not become empty, the function terminates. Otherwise, the function removes the process descriptor from the hash table of type `type`; finally, if the PID number does not occur in any other hash table, the function clears the corresponding bit in the PID bitmap, so that the number can be recycled.

`next_thread(task)`

> Returns the process descriptor address of the lightweight process that follows `task` in the hash table list of type `PIDTYPE_TGID`. Because the hash table list is circular, when applied to a conventional process the macro returns the descriptor address of the process itself.

How Processes Are Organized

The runqueue lists group all processes in a `TASK_RUNNING` state. When it comes to grouping processes in other states, the various states call for different types of treatment, with Linux opting for one of the choices shown in the following list.

- Processes in a `TASK_STOPPED`, `EXIT_ZOMBIE`, or `EXIT_DEAD` state are not linked in specific lists. There is no need to group processes in any of these three states, because stopped, zombie, and dead processes are accessed only via PID or via linked lists of the child processes for a particular parent.

- Processes in a `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state are subdivided into many classes, each of which corresponds to a specific event. In this case, the process state does not provide enough information to retrieve the process quickly, so it is necessary to introduce additional lists of processes. These are called *wait queues* and are discussed next.

Wait queues

Wait queues have several uses in the kernel, particularly for interrupt handling, process synchronization, and timing. Because these topics are discussed in later chapters, we'll just say here that a process must often wait for some event to occur, such as for a disk operation to terminate, a system resource to be released, or a fixed interval of time to elapse. Wait queues implement conditional waits on events: a process wishing to wait for a specific event places itself in the proper wait queue and relinquishes control. Therefore, a wait queue represents a set of sleeping processes, which are woken up by the kernel when some condition becomes true.

Wait queues are implemented as doubly linked lists whose elements include pointers to process descriptors. Each wait queue is identified by a *wait queue head*, a data structure of type wait_queue_head_t:

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

Because wait queues are modified by interrupt handlers as well as by major kernel functions, the doubly linked lists must be protected from concurrent accesses, which could induce unpredictable results (see Chapter 5). Synchronization is achieved by the lock spin lock in the wait queue head. The task_list field is the head of the list of waiting processes.

Elements of a wait queue list are of type wait_queue_t:

```
struct __wait_queue {
    unsigned int flags;
    struct task_struct * task;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;
```

Each element in the wait queue list represents a sleeping process, which is waiting for some event to occur; its descriptor address is stored in the task field. The task_list field contains the pointers that link this element to the list of processes waiting for the same event.

However, it is not always convenient to wake up *all* sleeping processes in a wait queue. For instance, if two or more processes are waiting for exclusive access to some resource to be released, it makes sense to wake up just one process in the wait queue. This process takes the resource, while the other processes continue to sleep. (This avoids a problem known as the "thundering herd," with which multiple processes are wakened only to race for a resource that can be accessed by one of them, with the result that remaining processes must once more be put back to sleep.)

Thus, there are two kinds of sleeping processes: *exclusive processes* (denoted by the value 1 in the flags field of the corresponding wait queue element) are selectively woken up by the kernel, while *nonexclusive processes* (denoted by the value 0 in the flags field) are always woken up by the kernel when the event occurs. A process waiting for a resource that can be granted to just one process at a time is a typical exclusive process. Processes waiting for an event that may concern any of them are nonexclusive. Consider, for instance, a group of processes that are waiting for the termination of a group of disk block transfers: as soon as the transfers complete, all waiting processes must be woken up. As we'll see next, the func field of a wait queue element is used to specify how the processes sleeping in the wait queue should be woken up.

Handling wait queues

A new wait queue head may be defined by using the `DECLARE_WAIT_QUEUE_HEAD(name)` macro, which statically declares a new wait queue head variable called `name` and initializes its `lock` and `task_list` fields. The `init_waitqueue_head()` function may be used to initialize a wait queue head variable that was allocated dynamically.

The `init_waitqueue_entry(q,p)` function initializes a `wait_queue_t` structure q as follows:

```
q->flags = 0;
q->task = p;
q->func = default_wake_function;
```

The nonexclusive process p will be awakened by `default_wake_function()`, which is a simple wrapper for the `try_to_wake_up()` function discussed in Chapter 7.

Alternatively, the `DEFINE_WAIT` macro declares a new `wait_queue_t` variable and initializes it with the descriptor of the process currently executing on the CPU and the address of the `autoremove_wake_function()` wake-up function. This function invokes `default_wake_function()` to awaken the sleeping process, and then removes the wait queue element from the wait queue list. Finally, a kernel developer can define a custom awakening function by initializing the wait queue element with the `init_waitqueue_func_entry()` function.

Once an element is defined, it must be inserted into a wait queue. The `add_wait_queue()` function inserts a nonexclusive process in the first position of a wait queue list. The `add_wait_queue_exclusive()` function inserts an exclusive process in the last position of a wait queue list. The `remove_wait_queue()` function removes a process from a wait queue list. The `waitqueue_active()` function checks whether a given wait queue list is empty.

A process wishing to wait for a specific condition can invoke any of the functions shown in the following list.

- The `sleep_on()` function operates on the current process:

    ```
    void sleep_on(wait_queue_head_t *wq)
    {
        wait_queue_t wait;
        init_waitqueue_entry(&wait, current);
        current->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue(wq,&wait); /*wq points to the wait queue head*/
        schedule();
        remove_wait_queue(wq, &wait);
    }
    ```

    The function sets the state of the current process to `TASK_UNINTERRUPTIBLE` and inserts it into the specified wait queue. Then it invokes the scheduler, which resumes the execution of another process. When the sleeping process is awakened, the scheduler resumes execution of the `sleep_on()` function, which removes the process from the wait queue.

- The `interruptible_sleep_on()` function is identical to `sleep_on()`, except that it sets the state of the current process to `TASK_INTERRUPTIBLE` instead of setting it to `TASK_UNINTERRUPTIBLE`, so that the process also can be woken up by receiving a signal.

- The `sleep_on_timeout()` and `interruptible_sleep_on_timeout()` functions are similar to the previous ones, but they also allow the caller to define a time interval after which the process will be woken up by the kernel. To do this, they invoke the `schedule_timeout()` function instead of `schedule()` (see the section "An Application of Dynamic Timers: the nanosleep() System Call" in Chapter 6).

- The `prepare_to_wait()`, `prepare_to_wait_exclusive()`, and `finish_wait()` functions, introduced in Linux 2.6, offer yet another way to put the current process to sleep in a wait queue. Typically, they are used as follows:

  ```
  DEFINE_WAIT(wait);
  prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);
                          /* wq is the head of the wait queue */
  ...
  if (!condition)
      schedule();
  finish_wait(&wq, &wait);
  ```

  The `prepare_to_wait()` and `prepare_to_wait_exclusive()` functions set the process state to the value passed as the third parameter, then set the exclusive flag in the wait queue element respectively to 0 (nonexclusive) or 1 (exclusive), and finally insert the wait queue element `wait` into the list of the wait queue head `wq`.

  As soon as the process is awakened, it executes the `finish_wait()` function, which sets again the process state to `TASK_RUNNING` (just in case the awaking condition becomes true before invoking `schedule()`), and removes the wait queue element from the wait queue list (unless this has already been done by the wakeup function).

- The `wait_event` and `wait_event_interruptible` macros put the calling process to sleep on a wait queue until a given condition is verified. For instance, the `wait_event(wq,condition)` macro essentially yields the following fragment:

  ```
  DEFINE_WAIT(__wait);
  for (;;) {
      prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
      if (condition)
          break;
      schedule();
  }
  finish_wait(&wq, &__wait);
  ```

A few comments on the functions mentioned in the above list: the `sleep_on()`-like functions cannot be used in the common situation where one has to test a condition and atomically put the process to sleep when the condition is not verified; therefore, because they are a well-known source of race conditions, their use is discouraged.

Moreover, in order to insert an exclusive process into a wait queue, the kernel must make use of the `prepare_to_wait_exclusive()` function (or just invoke `add_wait_queue_exclusive()` directly); any other helper function inserts the process as nonexclusive. Finally, unless `DEFINE_WAIT` or `finish_wait()` are used, the kernel must remove the wait queue element from the list after the waiting process has been awakened.

The kernel awakens processes in the wait queues, putting them in the `TASK_RUNNING` state, by means of one of the following macros: `wake_up`, `wake_up_nr`, `wake_up_all`, `wake_up_interruptible`, `wake_up_interruptible_nr`, `wake_up_interruptible_all`, `wake_up_interruptible_sync,` and `wake_up_locked`. One can understand what each of these nine macros does from its name:

- All macros take into consideration sleeping processes in the `TASK_INTERRUPTIBLE` state; if the macro name does not include the string "interruptible," sleeping processes in the `TASK_UNINTERRUPTIBLE` state also are considered.

- All macros wake all nonexclusive processes having the required state (see the previous bullet item).

- The macros whose name include the string "nr" wake a given number of exclusive processes having the required state; this number is a parameter of the macro. The macros whose names include the string "all" wake all exclusive processes having the required state. Finally, the macros whose names don't include "nr" or "all" wake exactly one exclusive process that has the required state.

- The macros whose names don't include the string "sync" check whether the priority of any of the woken processes is higher than that of the processes currently running in the systems and invoke `schedule()` if necessary. These checks are not made by the macro whose name includes the string "sync"; as a result, execution of a high priority process might be slightly delayed.

- The `wake_up_locked` macro is similar to `wake_up`, except that it is called when the spin lock in `wait_queue_head_t` is already held.

For instance, the `wake_up` macro is essentially equivalent to the following code fragment:

```
void wake_up(wait_queue_head_t *q)
{
    struct list_head *tmp;
    wait_queue_t *curr;

    list_for_each(tmp, &q->task_list) {
        curr = list_entry(tmp, wait_queue_t, task_list);
        if (curr->func(curr, TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
                    0, NULL) && curr->flags)
            break;
    }
}
```

The `list_for_each` macro scans all items in the `q->task_list` doubly linked list, that is, all processes in the wait queue. For each item, the `list_entry` macro computes the address of the corresponding `wait_queue_t` variable. The `func` field of this variable stores the address of the wake-up function, which tries to wake up the process identified by the `task` field of the wait queue element. If a process has been effectively awakened (the function returned 1) and if the process is exclusive (`curr->flags` equal to 1), the loop terminates. Because all nonexclusive processes are always at the beginning of the doubly linked list and all exclusive processes are at the end, the function always wakes the nonexclusive processes and then wakes one exclusive process, if any exists.[*]

Process Resource Limits

Each process has an associated set of *resource limits*, which specify the amount of system resources it can use. These limits keep a user from overwhelming the system (its CPU, disk space, and so on). Linux recognizes the following resource limits illustrated in Table 3-7.

The resource limits for the current process are stored in the `current->signal->rlim` field, that is, in a field of the process's signal descriptor (see the section "Data Structures Associated with Signals" in Chapter 11). The field is an array of elements of type `struct rlimit`, one for each resource limit:

```
struct rlimit {
    unsigned long rlim_cur;
    unsigned long rlim_max;
};
```

Table 3-7. Resource limits

Field name	Description
RLIMIT_AS	The maximum size of process address space, in bytes. The kernel checks this value when the process uses `malloc( )` or a related function to enlarge its address space (see the section "The Process's Address Space" in Chapter 9).
RLIMIT_CORE	The maximum core dump file size, in bytes. The kernel checks this value when a process is aborted, before creating a core file in the current directory of the process (see the section "Actions Performed upon Delivering a Signal" in Chapter 11). If the limit is 0, the kernel won't create the file.
RLIMIT_CPU	The maximum CPU time for the process, in seconds. If the process exceeds the limit, the kernel sends it a `SIGXCPU` signal, and then, if the process doesn't terminate, a `SIGKILL` signal (see Chapter 11).
RLIMIT_DATA	The maximum heap size, in bytes. The kernel checks this value before expanding the heap of the process (see the section "Managing the Heap" in Chapter 9).

---

[*] By the way, it is rather uncommon that a wait queue includes both exclusive and nonexclusive processes.

Table 3-7. Resource limits (continued)

Field name	Description
RLIMIT_FSIZE	The maximum file size allowed, in bytes. If the process tries to enlarge a file to a size greater than this value, the kernel sends it a SIGXFSZ signal.
RLIMIT_LOCKS	Maximum number of file locks (currently, not enforced).
RLIMIT_MEMLOCK	The maximum size of nonswappable memory, in bytes. The kernel checks this value when the process tries to lock a page frame in memory using the mlock( ) or mlockall( ) system calls (see the section "Allocating a Linear Address Interval" in Chapter 9).
RLIMIT_MSGQUEUE	Maximum number of bytes in POSIX message queues (see the section "POSIX Message Queues" in Chapter 19).
RLIMIT_NOFILE	The maximum number of open file descriptors. The kernel checks this value when opening a new file or duplicating a file descriptor (see Chapter 12).
RLIMIT_NPROC	The maximum number of processes that the user can own (see the section "The clone( ), fork( ), and vfork( ) System Calls" later in this chapter).
RLIMIT_RSS	The maximum number of page frames owned by the process (currently, not enforced).
RLIMIT_SIGPENDING	The maximum number of pending signals for the process (see Chapter 11).
RLIMIT_STACK	The maximum stack size, in bytes. The kernel checks this value before expanding the User Mode stack of the process (see the section "Page Fault Exception Handler" in Chapter 9).

The rlim_cur field is the current resource limit for the resource. For example, current->signal->rlim[RLIMIT_CPU].rlim_cur represents the current limit on the CPU time of the running process.

The rlim_max field is the maximum allowed value for the resource limit. By using the getrlimit( ) and setrlimit( ) system calls, a user can always increase the rlim_cur limit of some resource up to rlim_max. However, only the superuser (or, more precisely, a user who has the CAP_SYS_RESOURCE capability) can increase the rlim_max field or set the rlim_cur field to a value greater than the corresponding rlim_max field.

Most resource limits contain the value RLIM_INFINITY (0xffffffff), which means that no user limit is imposed on the corresponding resource (of course, real limits exist due to kernel design restrictions, available RAM, available space on disk, etc.). However, the system administrator may choose to impose stronger limits on some resources. Whenever a user logs into the system, the kernel creates a process owned by the superuser, which can invoke setrlimit( ) to decrease the rlim_max and rlim_cur fields for a resource. The same process later executes a login shell and becomes owned by the user. Each new process created by the user inherits the content of the rlim array from its parent, and therefore the user cannot override the limits enforced by the administrator.

Process Switch

To control the execution of processes, the kernel must be able to suspend the execution of the process running on the CPU and resume the execution of some other process previously suspended. This activity goes variously by the names *process switch*,

task switch, or *context switch*. The next sections describe the elements of process switching in Linux.

Hardware Context

While each process can have its own address space, all processes have to share the CPU registers. So before resuming the execution of a process, the kernel must ensure that each such register is loaded with the value it had when the process was suspended.

The set of data that must be loaded into the registers before the process resumes its execution on the CPU is called the *hardware context*. The hardware context is a subset of the process execution context, which includes all information needed for the process execution. In Linux, a part of the hardware context of a process is stored in the process descriptor, while the remaining part is saved in the Kernel Mode stack.

In the description that follows, we will assume the prev local variable refers to the process descriptor of the process being switched out and next refers to the one being switched in to replace it. We can thus define a *process switch* as the activity consisting of saving the hardware context of prev and replacing it with the hardware context of next. Because process switches occur quite often, it is important to minimize the time spent in saving and loading hardware contexts.

Old versions of Linux took advantage of the hardware support offered by the 80×86 architecture and performed a process switch through a far jmp instruction* to the selector of the Task State Segment Descriptor of the next process. While executing the instruction, the CPU performs a *hardware context switch* by automatically saving the old hardware context and loading a new one. But Linux 2.6 uses software to perform a process switch for the following reasons:

- Step-by-step switching performed through a sequence of mov instructions allows better control over the validity of the data being loaded. In particular, it is possible to check the values of the ds and es segmentation registers, which might have been forged by a malicious user. This type of checking is not possible when using a single far jmp instruction.

- The amount of time required by the old approach and the new approach is about the same. However, it is not possible to optimize a hardware context switch, while there might be room for improving the current switching code.

Process switching occurs only in Kernel Mode. The contents of all registers used by a process in User Mode have already been saved on the Kernel Mode stack before performing process switching (see Chapter 4). This includes the contents of the ss and esp pair that specifies the User Mode stack pointer address.

---

* far jmp instructions modify both the cs and eip registers, while simple jmp instructions modify only eip.

Task State Segment

The 80×86 architecture includes a specific segment type called the *Task State Segment* (TSS), to store hardware contexts. Although Linux doesn't use hardware context switches, it is nonetheless forced to set up a TSS for each distinct CPU in the system. This is done for two main reasons:

- When an 80×86 CPU switches from User Mode to Kernel Mode, it fetches the address of the Kernel Mode stack from the TSS (see the sections "Hardware Handling of Interrupts and Exceptions" in Chapter 4 and "Issuing a System Call via the sysenter Instruction" in Chapter 10).

- When a User Mode process attempts to access an I/O port by means of an `in` or `out` instruction, the CPU may need to access an I/O Permission Bitmap stored in the TSS to verify whether the process is allowed to address the port.

    More precisely, when a process executes an `in` or `out` I/O instruction in User Mode, the control unit performs the following operations:

    1. It checks the 2-bit IOPL field in the `eflags` register. If it is set to 3, the control unit executes the I/O instructions. Otherwise, it performs the next check.

    2. It accesses the `tr` register to determine the current TSS, and thus the proper I/O Permission Bitmap.

    3. It checks the bit of the I/O Permission Bitmap corresponding to the I/O port specified in the I/O instruction. If it is cleared, the instruction is executed; otherwise, the control unit raises a "General protection" exception.

The `tss_struct` structure describes the format of the TSS. As already mentioned in Chapter 2, the `init_tss` array stores one TSS for each CPU on the system. At each process switch, the kernel updates some fields of the TSS so that the corresponding CPU's control unit may safely retrieve the information it needs. Thus, the TSS reflects the privilege of the current process on the CPU, but there is no need to maintain TSSs for processes when they're not running.

Each TSS has its own 8-byte *Task State Segment Descriptor* (TSSD). This descriptor includes a 32-bit `Base` field that points to the TSS starting address and a 20-bit `Limit` field. The `S` flag of a TSSD is cleared to denote the fact that the corresponding TSS is a System Segment (see the section "Segment Descriptors" in Chapter 2).

The `Type` field is set to either 9 or 11 to denote that the segment is actually a TSS. In the Intel's original design, each process in the system should refer to its own TSS; the second least significant bit of the `Type` field is called the *Busy bit*; it is set to 1 if the process is being executed by a CPU, and to 0 otherwise. In Linux design, there is just one TSS for each CPU, so the Busy bit is always set to 1.

The TSSDs created by Linux are stored in the Global Descriptor Table (GDT), whose base address is stored in the `gdtr` register of each CPU. The `tr` register of each CPU

contains the TSSD Selector of the corresponding TSS. The register also includes two hidden, nonprogrammable fields: the Base and Limit fields of the TSSD. In this way, the processor can address the TSS directly without having to retrieve the TSS address from the GDT.

The thread field

At every process switch, the hardware context of the process being replaced must be saved somewhere. It cannot be saved on the TSS, as in the original Intel design, because Linux uses a single TSS for each processor, instead of one for every process.

Thus, each process descriptor includes a field called thread of type thread_struct, in which the kernel saves the hardware context whenever the process is being switched out. As we'll see later, this data structure includes fields for most of the CPU registers, except the general-purpose registers such as eax, ebx, etc., which are stored in the Kernel Mode stack.

Performing the Process Switch

A process switch may occur at just one well-defined point: the schedule() function, which is discussed at length in Chapter 7. Here, we are only concerned with how the kernel performs a process switch.

Essentially, every process switch consists of two steps:

1. Switching the Page Global Directory to install a new address space; we'll describe this step in Chapter 9.
2. Switching the Kernel Mode stack and the hardware context, which provides all the information needed by the kernel to execute the new process, including the CPU registers.

Again, we assume that prev points to the descriptor of the process being replaced, and next to the descriptor of the process being activated. As we'll see in Chapter 7, prev and next are local variables of the schedule() function.

The switch_to macro

The second step of the process switch is performed by the switch_to macro. It is one of the most hardware-dependent routines of the kernel, and it takes some effort to understand what it does.

First of all, the macro has three parameters, called prev, next, and last. You might easily guess the role of prev and next: they are just placeholders for the local variables prev and next, that is, they are input parameters that specify the memory locations containing the descriptor address of the process being replaced and the descriptor address of the new process, respectively.

What about the third parameter, `last`? Well, in any process switch three processes are involved, not just two. Suppose the kernel decides to switch off process A and to activate process B. In the `schedule( )` function, `prev` points to A's descriptor and `next` points to B's descriptor. As soon as the `switch_to` macro deactivates A, the execution flow of A freezes.

Later, when the kernel wants to reactivate A, it must switch off another process C (in general, this is different from B) by executing another `switch_to` macro with `prev` pointing to C and `next` pointing to A. When A resumes its execution flow, it finds its old Kernel Mode stack, so the `prev` local variable points to A's descriptor and `next` points to B's descriptor. The scheduler, which is now executing on behalf of process A, has lost any reference to C. This reference, however, turns out to be useful to complete the process switching (see Chapter 7 for more details).

The last parameter of the `switch_to` macro is an output parameter that specifies a memory location in which the macro writes the descriptor address of process C (of course, this is done after A resumes its execution). Before the process switching, the macro saves in the eax CPU register the content of the variable identified by the first input parameter `prev`—that is, the `prev` local variable allocated on the Kernel Mode stack of A. After the process switching, when A has resumed its execution, the macro writes the content of the eax CPU register in the memory location of A identified by the third output parameter `last`. Because the CPU register doesn't change across the process switch, this memory location receives the address of C's descriptor. In the current implementation of `schedule( )`, the last parameter identifies the `prev` local variable of A, so `prev` is overwritten with the address of C.

The contents of the Kernel Mode stacks of processes A, B, and C are shown in Figure 3-7, together with the values of the eax register; be warned that the figure shows the value of the `prev` local variable *before* its value is overwritten with the contents of the eax register.
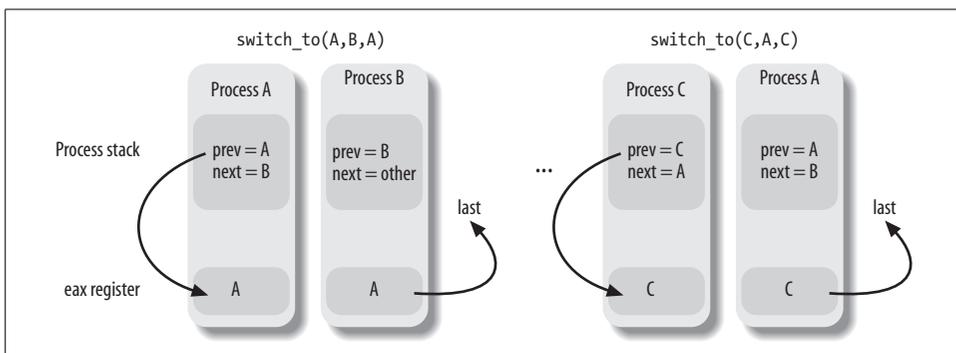
Figure 3-7. Preserving the reference to process C across a process switch

The `switch_to` macro is coded in *extended inline assembly language* that makes for rather complex reading: in fact, the code refers to registers by means of a special

positional notation that allows the compiler to freely choose the general-purpose registers to be used. Rather than follow the cumbersome extended inline assembly language, we'll describe what the switch_to macro typically does on an 80×86 microprocessor by using standard assembly language:

1. Saves the values of prev and next in the eax and edx registers, respectively:
   ```
   movl prev, %eax
   movl next, %edx
   ```

2. Saves the contents of the eflags and ebp registers in the prev Kernel Mode stack. They must be saved because the compiler assumes that they will stay unchanged until the end of switch_to:
   ```
   pushfl
   pushl %ebp
   ```

3. Saves the content of esp in prev->thread.esp so that the field points to the top of the prev Kernel Mode stack:
   ```
   movl %esp,484(%eax)
   ```
   The 484(%eax) operand identifies the memory cell whose address is the contents of eax plus 484.

4. Loads next->thread.esp in esp. From now on, the kernel operates on the Kernel Mode stack of next, so this instruction performs the actual process switch from prev to next. Because the address of a process descriptor is closely related to that of the Kernel Mode stack (as explained in the section "Identifying a Process" earlier in this chapter), changing the kernel stack means changing the current process:
   ```
   movl 484(%edx), %esp
   ```

5. Saves the address labeled 1 (shown later in this section) in prev->thread.eip. When the process being replaced resumes its execution, the process executes the instruction labeled as 1:
   ```
   movl $1f, 480(%eax)
   ```

6. On the Kernel Mode stack of next, the macro pushes the next->thread.eip value, which, in most cases, is the address labeled as 1:
   ```
   pushl 480(%edx)
   ```

7. Jumps to the __switch_to( ) C function (see next):
   ```
   jmp __switch_to
   ```

8. Here process A that was replaced by B gets the CPU again: it executes a few instructions that restore the contents of the eflags and ebp registers. The first of these two instructions is labeled as 1:
   ```
   1:
       popl %ebp
       popfl
   ```

Notice how these pop instructions refer to the kernel stack of the prev process. They will be executed when the scheduler selects prev as the new process to be

executed on the CPU, thus invoking switch_to with prev as the second parameter. Therefore, the esp register points to the prev's Kernel Mode stack.

9. Copies the content of the eax register (loaded in step 1 above) into the memory location identified by the third parameter last of the switch_to macro:

```
movl %eax, last
```

As discussed earlier, the eax register points to the descriptor of the process that has just been replaced.*

The __switch_to( ) function

The __switch_to( ) function does the bulk of the process switch started by the switch_to( ) macro. It acts on the prev_p and next_p parameters that denote the former process and the new process. This function call is different from the average function call, though, because __switch_to( ) takes the prev_p and next_p parameters from the eax and edx registers (where we saw they were stored), not from the stack like most functions. To force the function to go to the registers for its parameters, the kernel uses the __attribute__ and regparm keywords, which are nonstandard extensions of the C language implemented by the gcc compiler. The __switch_to( ) function is declared in the *include/asm-i386/system.h* header file as follows:

```
__switch_to(struct task_struct *prev_p,
            struct task_struct *next_p)
    __attribute__(regparm(3));
```

The steps performed by the function are the following:

1. Executes the code yielded by the __unlazy_fpu( ) macro (see the section "Saving and Loading the FPU, MMX, and XMM Registers" later in this chapter) to optionally save the contents of the FPU, MMX, and XMM registers of the prev_p process.

```
__unlazy_fpu(prev_p);
```

2. Executes the smp_processor_id( ) macro to get the index of the *local CPU*, namely the CPU that executes the code. The macro gets the index from the cpu field of the thread_info structure of the current process and stores it into the cpu local variable.

3. Loads next_p->thread.esp0 in the esp0 field of the TSS relative to the local CPU; as we'll see in the section "Issuing a System Call via the sysenter Instruction" in Chapter 10, any future privilege level change from User Mode to Kernel Mode raised by a sysenter assembly instruction will copy this address in the esp register:

```
init_tss[cpu].esp0 = next_p->thread.esp0;
```

---

* As stated earlier in this section, the current implementation of the schedule( ) function reuses the prev local variable, so that the assembly language instruction looks like movl %eax,prev.

4. Loads in the Global Descriptor Table of the local CPU the Thread-Local Storage (TLS) segments used by the `next_p` process; the three Segment Selectors are stored in the `tls_array` array inside the process descriptor (see the section "Segmentation in Linux" in Chapter 2).

```
cpu_gdt_table[cpu][6] = next_p->thread.tls_array[0];
cpu_gdt_table[cpu][7] = next_p->thread.tls_array[1];
cpu_gdt_table[cpu][8] = next_p->thread.tls_array[2];
```

5. Stores the contents of the `fs` and `gs` segmentation registers in `prev_p->thread.fs` and `prev_p->thread.gs`, respectively; the corresponding assembly language instructions are:

```
movl %fs, 40(%esi)
movl %gs, 44(%esi)
```

The `esi` register points to the `prev_p->thread` structure.

6. If the `fs` or the `gs` segmentation register have been used either by the `prev_p` or by the `next_p` process (i.e., if they have a nonzero value), loads into these registers the values stored in the `thread_struct` descriptor of the `next_p` process. This step logically complements the actions performed in the previous step. The main assembly language instructions are:

```
movl 40(%ebx),%fs
movl 44(%ebx),%gs
```

The `ebx` register points to the `next_p->thread` structure. The code is actually more intricate, as an exception might be raised by the CPU when it detects an invalid segment register value. The code takes this possibility into account by adopting a "fix-up" approach (see the section "Dynamic Address Checking: The Fix-up Code" in Chapter 10).

7. Loads six of the `dr0`, …, `dr7` debug registers[*] with the contents of the `next_p->thread.debugreg` array. This is done only if `next_p` was using the debug registers when it was suspended (that is, field `next_p->thread.debugreg[7]` is not 0). These registers need not be saved, because the `prev_p->thread.debugreg` array is modified only when a debugger wants to monitor `prev`:

```
if (next_p->thread.debugreg[7]){
    loaddebug(&next_p->thread, 0);
    loaddebug(&next_p->thread, 1);
    loaddebug(&next_p->thread, 2);
    loaddebug(&next_p->thread, 3);
    /* no 4 and 5 */
    loaddebug(&next_p->thread, 6);
    loaddebug(&next_p->thread, 7);
}
```

---

[*] The 80×86 debug registers allow a process to be monitored by the hardware. Up to four breakpoint areas may be defined. Whenever a monitored process issues a linear address included in one of the breakpoint areas, an exception occurs.

8. Updates the I/O bitmap in the TSS, if necessary. This must be done when either next_p or prev_p has its own customized I/O Permission Bitmap:

```
if (prev_p->thread.io_bitmap_ptr || next_p->thread.io_bitmap_ptr)
    handle_io_bitmap(&next_p->thread, &init_tss[cpu]);
```

Because processes seldom modify the I/O Permission Bitmap, this bitmap is handled in a "lazy" mode: the actual bitmap is copied into the TSS of the local CPU only if a process actually accesses an I/O port in the current timeslice. The customized I/O Permission Bitmap of a process is stored in a buffer pointed to by the io_bitmap_ptr field of the thread_info structure. The handle_io_bitmap() function sets up the io_bitmap field of the TSS used by the local CPU for the next_p process as follows:

- If the next_p process does not have its own customized I/O Permission Bitmap, the io_bitmap field of the TSS is set to the value 0x8000.

- If the next_p process has its own customized I/O Permission Bitmap, the io_bitmap field of the TSS is set to the value 0x9000.

The io_bitmap field of the TSS should contain an offset inside the TSS where the actual bitmap is stored. The 0x8000 and 0x9000 values point outside of the TSS limit and will thus cause a "General protection" exception whenever the User Mode process attempts to access an I/O port (see the section "Exceptions" in Chapter 4). The do_general_protection() exception handler will check the value stored in the io_bitmap field: if it is 0x8000, the function sends a SIGSEGV signal to the User Mode process; otherwise, if it is 0x9000, the function copies the process bitmap (pointed to by the io_bitmap_ptr field in the thread_info structure) in the TSS of the local CPU, sets the io_bitmap field to the actual bitmap offset (104), and forces a new execution of the faulty assembly language instruction.

9. Terminates. The __switch_to() C function ends by means of the statement:

```
return prev_p;
```

The corresponding assembly language instructions generated by the compiler are:

```
movl %edi,%eax
ret
```

The prev_p parameter (now in edi) is copied into eax, because by default the return value of any C function is passed in the eax register. Notice that the value of eax is thus preserved across the invocation of __switch_to(); this is quite important, because the invoking switch_to macro assumes that eax always stores the address of the process descriptor being replaced.

The ret assembly language instruction loads the eip program counter with the return address stored on top of the stack. However, the __switch_to() function has been invoked simply by jumping into it. Therefore, the ret instruction finds on the stack the address of the instruction labeled as 1, which was pushed by the switch_to macro. If next_p was never suspended before because it is being

executed for the first time, the function finds the starting address of the `ret_from_fork()` function (see the section "The clone(), fork(), and vfork() System Calls" later in this chapter).

Saving and Loading the FPU, MMX, and XMM Registers

Starting with the Intel 80486DX, the arithmetic floating-point unit (FPU) has been integrated into the CPU. The name *mathematical coprocessor* continues to be used in memory of the days when floating-point computations were executed by an expensive special-purpose chip. To maintain compatibility with older models, however, floating-point arithmetic functions are performed with *ESCAPE instructions*, which are instructions with a prefix byte ranging between `0xd8` and `0xdf`. These instructions act on the set of floating-point registers included in the CPU. Clearly, if a process is using ESCAPE instructions, the contents of the floating-point registers belong to its hardware context and should be saved.

In later Pentium models, Intel introduced a new set of assembly language instructions into its microprocessors. They are called *MMX instructions* and are supposed to speed up the execution of multimedia applications. MMX instructions act on the floating-point registers of the FPU. The obvious disadvantage of this architectural choice is that programmers cannot mix floating-point instructions and MMX instructions. The advantage is that operating system designers can ignore the new instruction set, because the same facility of the task-switching code for saving the state of the floating-point unit can also be relied upon to save the MMX state.

MMX instructions speed up multimedia applications, because they introduce a single-instruction multiple-data (SIMD) pipeline inside the processor. The Pentium III model extends that SIMD capability: it introduces the *SSE extensions* (Streaming SIMD Extensions), which adds facilities for handling floating-point values contained in eight 128-bit registers called the XMM registers. Such registers do not overlap with the FPU and MMX registers, so SSE and FPU/MMX instructions may be freely mixed. The Pentium 4 model introduces yet another feature: the *SSE2 extensions*, which is basically an extension of SSE supporting higher-precision floating-point values. SSE2 uses the same set of XMM registers as SSE.

The 80×86 microprocessors do not automatically save the FPU, MMX, and XMM registers in the TSS. However, they include some hardware support that enables kernels to save these registers only when needed. The hardware support consists of a `TS` (Task-Switching) flag in the `cr0` register, which obeys the following rules:

- Every time a hardware context switch is performed, the `TS` flag is set.
- Every time an ESCAPE, MMX, SSE, or SSE2 instruction is executed when the `TS` flag is set, the control unit raises a "Device not available" exception (see Chapter 4).

The TS flag allows the kernel to save and restore the FPU, MMX, and XMM registers only when really needed. To illustrate how it works, suppose that a process A is using the mathematical coprocessor. When a context switch occurs from A to B, the kernel sets the TS flag and saves the floating-point registers into the TSS of process A. If the new process B does not use the mathematical coprocessor, the kernel won't need to restore the contents of the floating-point registers. But as soon as B tries to execute an ESCAPE or MMX instruction, the CPU raises a "Device not available" exception, and the corresponding handler loads the floating-point registers with the values saved in the TSS of process B.

Let's now describe the data structures introduced to handle selective loading of the FPU, MMX, and XMM registers. They are stored in the thread.i387 subfield of the process descriptor, whose format is described by the i387_union union:

```
union i387_union {
    struct i387_fsave_struct    fsave;
    struct i387_fxsave_struct   fxsave;
    struct i387_soft_struct     soft;
};
```

As you see, the field may store just one of three different types of data structures. The i387_soft_struct type is used by CPU models without a mathematical coprocessor; the Linux kernel still supports these old chips by emulating the coprocessor via software. We don't discuss this legacy case further, however. The i387_fsave_struct type is used by CPU models with a mathematical coprocessor and, optionally, an MMX unit. Finally, the i387_fxsave_struct type is used by CPU models featuring SSE and SSE2 extensions.

The process descriptor includes two additional flags:

- The TS_USEDFPU flag, which is included in the status field of the thread_info descriptor. It specifies whether the process used the FPU, MMX, or XMM registers in the current execution run.

- The PF_USED_MATH flag, which is included in the flags field of the task_struct descriptor. This flag specifies whether the contents of the thread.i387 subfield are significant. The flag is cleared (not significant) in two cases, shown in the following list.

  — When the process starts executing a new program by invoking an execve( ) system call (see Chapter 20). Because control will never return to the former program, the data currently stored in thread.i387 is never used again.

  — When a process that was executing a program in User Mode starts executing a signal handler procedure (see Chapter 11). Because signal handlers are asynchronous with respect to the program execution flow, the floating-point registers could be meaningless to the signal handler. However, the kernel saves the floating-point registers in thread.i387 before starting the handler and restores them after the handler terminates. Therefore, a signal handler is allowed to use the mathematical coprocessor.

Saving the FPU registers

As stated earlier, the `__switch_to()` function executes the `__unlazy_fpu` macro, passing the process descriptor of the `prev` process being replaced as an argument. The macro checks the value of the `TS_USEDFPU` flags of `prev`. If the flag is set, `prev` has used an FPU, MMX, SSE, or SSE2 instructions; therefore, the kernel must save the relative hardware context:

```
if (prev->thread_info->status & TS_USEDFPU)
    save_init_fpu(prev);
```

The `save_init_fpu()` function, in turn, executes essentially the following operations:

1. Dumps the contents of the FPU registers in the process descriptor of `prev` and then reinitializes the FPU. If the CPU uses SSE/SSE2 extensions, it also dumps the contents of the XMM registers and reinitializes the SSE/SSE2 unit. A couple of powerful extended inline assembly language instructions take care of everything, either:

```
asm volatile( "fxsave %0 ; fnclex"
    : "=m" (prev->thread.i387.fxsave) );
```

   if the CPU uses SSE/SSE2 extensions, or otherwise:

```
asm volatile( "fnsave %0 ; fwait"
    : "=m" (prev->thread.i387.fsave) );
```

2. Resets the `TS_USEDFPU` flag of `prev`:

```
prev->thread_info->status &= ~TS_USEDFPU;
```

3. Sets the TS flag of `cr0` by means of the `stts()` macro, which in practice yields assembly language instructions like the following:

```
movl %cr0, %eax
orl $8,%eax
movl %eax, %cr0
```

Loading the FPU registers

The contents of the floating-point registers are not restored right after the `next` process resumes execution. However, the TS flag of `cr0` has been set by `__unlazy_fpu()`. Thus, the first time the `next` process tries to execute an ESCAPE, MMX, or SSE/SSE2 instruction, the control unit raises a "Device not available" exception, and the kernel (more precisely, the exception handler involved by the exception) runs the `math_state_restore()` function. The `next` process is identified by this handler as `current`.

```
void math_state_restore()
{
    asm volatile ("clts"); /* clear the TS flag of cr0 */
    if (!(current->flags & PF_USED_MATH))
        init_fpu(current);
    restore_fpu(current);
    current->thread.status |= TS_USEDFPU;
}
```

The function clears the TC flags of `cr0`, so that further FPU, MMX, or SSE/SSE2 instructions executed by the process won't trigger the "Device not available" exception. If the contents of the `thread.i387` subfield are not significant, i.e., if the `PF_USED_MATH` flag is equal to 0, `init_fpu()` is invoked to reset the `thread.i387` subfield and to set the `PF_USED_MATH` flag of `current` to 1. The `restore_fpu()` function is then invoked to load the FPU registers with the proper values stored in the `thread.i387` subfield. To do this, either the `fxrstor` or the `frstor` assembly language instructions are used, depending on whether the CPU supports SSE/SSE2 extensions. Finally, `math_state_restore()` sets the `TS_USEDFPU` flag.

Using the FPU, MMX, and SSE/SSE2 units in Kernel Mode

Even the kernel can make use of the FPU, MMX, or SSE/SSE2 units. In doing so, of course, it should avoid interfering with any computation carried on by the current User Mode process. Therefore:

- Before using the coprocessor, the kernel must invoke `kernel_fpu_begin()`, which essentially calls `save_init_fpu()` to save the contents of the registers if the User Mode process used the FPU (`TS_USEDFPU` flag), and then resets the `TS` flag of the `cr0` register.

- After using the coprocessor, the kernel must invoke `kernel_fpu_end()`, which sets the `TS` flag of the `cr0` register.

Later, when the User Mode process executes a coprocessor instruction, the `math_state_restore()` function will restore the contents of the registers, just as in process switch handling.

It should be noted, however, that the execution time of `kernel_fpu_begin()` is rather large when the current User Mode process is using the coprocessor, so much as to nullify the speedup obtained by using the FPU, MMX, or SSE/SSE2 units. As a matter of fact, the kernel uses them only in a few places, typically when moving or clearing large memory areas or when computing checksum functions.

Creating Processes

Unix operating systems rely heavily on process creation to satisfy user requests. For example, the shell creates a new process that executes another copy of the shell whenever the user enters a command.

Traditional Unix systems treat all processes in the same way: resources owned by the parent process are duplicated in the child process. This approach makes process creation very slow and inefficient, because it requires copying the entire address space of the parent process. The child process rarely needs to read or modify all the resources inherited from the parent; in many cases, it issues an immediate `execve()` and wipes out the address space that was so carefully copied.

Modern Unix kernels solve this problem by introducing three different mechanisms:

- The Copy On Write technique allows both the parent and the child to read the same physical pages. Whenever either one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process. The implementation of this technique in Linux is fully explained in Chapter 9.

- Lightweight processes allow both the parent and the child to share many per-process kernel data structures, such as the paging tables (and therefore the entire User Mode address space), the open file tables, and the signal dispositions.

- The vfork( ) system call creates a process that shares the memory address space of its parent. To prevent the parent from overwriting data needed by the child, the parent's execution is blocked until the child exits or executes a new program. We'll learn more about the vfork( ) system call in the following section.

The clone( ), fork( ), and vfork( ) System Calls

Lightweight processes are created in Linux by using a function named clone( ), which uses the following parameters:

fn
> Specifies a function to be executed by the new process; when the function returns, the child terminates. The function returns an integer, which represents the exit code for the child process.

arg
> Points to data passed to the fn( ) function.

flags
> Miscellaneous information. The low byte specifies the signal number to be sent to the parent process when the child terminates; the SIGCHLD signal is generally selected. The remaining three bytes encode a group of clone flags, which are shown in Table 3-8.

child_stack
> Specifies the User Mode stack pointer to be assigned to the esp register of the child process. The invoking process (the parent) should always allocate a new stack for the child.

tls
> Specifies the address of a data structure that defines a Thread Local Storage segment for the new lightweight process (see the section "The Linux GDT" in Chapter 2). Meaningful only if the CLONE_SETTLS flag is set.

ptid
> Specifies the address of a User Mode variable of the parent process that will hold the PID of the new lightweight process. Meaningful only if the CLONE_PARENT_SETTID flag is set.

ctid

> Specifies the address of a User Mode variable of the new lightweight process that will hold the PID of such process. Meaningful only if the CLONE_CHILD_SETTID flag is set.

Table 3-8. Clone flags

Flag name	Description
CLONE_VM	Shares the memory descriptor and all Page Tables (see Chapter 9).
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bitmask used to mask the initial file permissions of a new file (the so-called file umask).
CLONE_FILES	Shares the table that identifies the open files (see Chapter 12).
CLONE_SIGHAND	Shares the tables that identify the signal handlers and the blocked and pending signals (see Chapter 11). If this flag is true, the CLONE_VM flag must also be set.
CLONE_PTRACE	If traced, the parent wants the child to be traced too. Furthermore, the debugger may want to trace the child on its own; in this case, the kernel forces the flag to 1.
CLONE_VFORK	Set when the system call issued is a vfork( ) (see later in this section).
CLONE_PARENT	Sets the parent of the child (parent and real_parent fields in the process descriptor) to the parent of the calling process.
CLONE_THREAD	Inserts the child into the same thread group of the parent, and forces the child to share the signal descriptor of the parent. The child's tgid and group_leader fields are set accordingly. If this flag is true, the CLONE_SIGHAND flag must also be set.
CLONE_NEWNS	Set if the clone needs its own namespace, that is, its own view of the mounted filesystems (see Chapter 12); it is not possible to specify both CLONE_NEWNS and CLONE_FS.
CLONE_SYSVSEM	Shares the System V IPC undoable semaphore operations (see the section "IPC Semaphores" in Chapter 19).
CLONE_SETTLS	Creates a new Thread Local Storage (TLS) segment for the lightweight process; the segment is described in the structure pointed to by the tls parameter.
CLONE_PARENT_SETTID	Writes the PID of the child into the User Mode variable of the parent pointed to by the ptid parameter.
CLONE_CHILD_CLEARTID	When set, the kernel sets up a mechanism to be triggered when the child process will exit or when it will start executing a new program. In these cases, the kernel will clear the User Mode variable pointed to by the ctid parameter and will awaken any process waiting for this event.
CLONE_DETACHED	A legacy flag ignored by the kernel.
CLONE_UNTRACED	Set by the kernel to override the value of the CLONE_PTRACE flag (used for disabling tracing of kernel threads; see the section "Kernel Threads" later in this chapter).
CLONE_CHILD_SETTID	Writes the PID of the child into the User Mode variable of the child pointed to by the ctid parameter.
CLONE_STOPPED	Forces the child to start in the TASK_STOPPED state.

clone( ) is actually a wrapper function defined in the C library (see the section "POSIX APIs and System Calls" in Chapter 10), which sets up the stack of the new

lightweight process and invokes a `clone()` system call hidden to the programmer. The `sys_clone()` service routine that implements the `clone()` system call does not have the `fn` and `arg` parameters. In fact, the wrapper function saves the pointer `fn` into the child's stack position corresponding to the return address of the wrapper function itself; the pointer `arg` is saved on the child's stack right below `fn`. When the wrapper function terminates, the CPU fetches the return address from the stack and executes the `fn(arg)` function.

The traditional `fork()` system call is implemented by Linux as a `clone()` system call whose `flags` parameter specifies both a `SIGCHLD` signal and all the clone flags cleared, and whose `child_stack` parameter is the current parent stack pointer. Therefore, the parent and child temporarily share the same User Mode stack. But thanks to the Copy On Write mechanism, they usually get separate copies of the User Mode stack as soon as one tries to change the stack.

The `vfork()` system call, introduced in the previous section, is implemented by Linux as a `clone()` system call whose `flags` parameter specifies both a `SIGCHLD` signal and the flags `CLONE_VM` and `CLONE_VFORK`, and whose `child_stack` parameter is equal to the current parent stack pointer.

The do_fork( ) function

The `do_fork()` function, which handles the `clone()`, `fork()`, and `vfork()` system calls, acts on the following parameters:

`clone_flags`
    Same as the `flags` parameter of `clone()`

`stack_start`
    Same as the `child_stack` parameter of `clone()`

`regs`
    Pointer to the values of the general purpose registers saved into the Kernel Mode stack when switching from User Mode to Kernel Mode (see the section "The do_IRQ( ) function" in Chapter 4)

`stack_size`
    Unused (always set to 0)

`parent_tidptr, child_tidptr`
    Same as the corresponding `ptid` and `ctid` parameters of `clone()`

`do_fork()` makes use of an auxiliary function called `copy_process()` to set up the process descriptor and any other kernel data structure required for child's execution. Here are the main steps performed by `do_fork()`:

1. Allocates a new PID for the child by looking in the `pidmap_array` bitmap (see the earlier section "Identifying a Process").

2. Checks the `ptrace` field of the parent (`current->ptrace`): if it is not zero, the parent process is being traced by another process, thus `do_fork()` checks whether

the debugger wants to trace the child on its own (independently of the value of the `CLONE_PTRACE` flag specified by the parent); in this case, if the child is not a kernel thread (`CLONE_UNTRACED` flag cleared), the function sets the `CLONE_PTRACE` flag.

3. Invokes `copy_process()` to make a copy of the process descriptor. If all needed resources are available, this function returns the address of the `task_struct` descriptor just created. This is the workhorse of the forking procedure, and we will describe it right after `do_fork()`.

4. If either the `CLONE_STOPPED` flag is set or the child process must be traced, that is, the `PT_PTRACED` flag is set in `p->ptrace`, it sets the state of the child to `TASK_STOPPED` and adds a pending `SIGSTOP` signal to it (see the section "The Role of Signals" in Chapter 11). The state of the child will remain `TASK_STOPPED` until another process (presumably the tracing process or the parent) will revert its state to `TASK_RUNNING`, usually by means of a `SIGCONT` signal.

5. If the `CLONE_STOPPED` flag is not set, it invokes the `wake_up_new_task()` function, which performs the following operations:

   a. Adjusts the scheduling parameters of both the parent and the child (see "The Scheduling Algorithm" in Chapter 7).

   b. If the child will run on the same CPU as the parent,[*] and parent and child do not share the same set of page tables (`CLONE_VM` flag cleared), it then forces the child to run before the parent by inserting it into the parent's runqueue right before the parent. This simple step yields better performance if the child flushes its address space and executes a new program right after the forking. If we let the parent run first, the Copy On Write mechanism would give rise to a series of unnecessary page duplications.

   c. Otherwise, if the child will not be run on the same CPU as the parent, or if parent and child share the same set of page tables (`CLONE_VM` flag set), it inserts the child in the last position of the parent's runqueue.

6. If the `CLONE_STOPPED` flag is set, it puts the child in the `TASK_STOPPED` state.

7. If the parent process is being traced, it stores the PID of the child in the `ptrace_message` field of `current` and invokes `ptrace_notify()`, which essentially stops the current process and sends a `SIGCHLD` signal to its parent. The "grandparent" of the child is the debugger that is tracing the parent; the `SIGCHLD` signal notifies the debugger that `current` has forked a child, whose PID can be retrieved by looking into the `current->ptrace_message` field.

8. If the `CLONE_VFORK` flag is specified, it inserts the parent process in a wait queue and suspends it until the child releases its memory address space (that is, until the child either terminates or executes a new program).

9. Terminates by returning the PID of the child.

---

[*] The parent process might be moved on to another CPU while the kernel forks the new process.

The copy_process() function

The copy_process() function sets up the process descriptor and any other kernel data structure required for a child's execution. Its parameters are the same as do_fork(), plus the PID of the child. Here is a description of its most significant steps:

1. Checks whether the flags passed in the clone_flags parameter are compatible. In particular, it returns an error code in the following cases:

   a. Both the flags CLONE_NEWNS and CLONE_FS are set.

   b. The CLONE_THREAD flag is set, but the CLONE_SIGHAND flag is cleared (lightweight processes in the same thread group must share signals).

   c. The CLONE_SIGHAND flag is set, but the CLONE_VM flag is cleared (lightweight processes sharing the signal handlers must also share the memory descriptor).

2. Performs any additional security checks by invoking security_task_create() and, later, security_task_alloc(). The Linux kernel 2.6 offers hooks for security extensions that enforce a security model stronger than the one adopted by traditional Unix. See Chapter 20 for details.

3. Invokes dup_task_struct() to get the process descriptor for the child. This function performs the following actions:

   a. Invokes __unlazy_fpu() on the current process to save, if necessary, the contents of the FPU, MMX, and SSE/SSE2 registers in the thread_info structure of the parent. Later, dup_task_struct() will copy these values in the thread_info structure of the child.

   b. Executes the alloc_task_struct() macro to get a process descriptor (task_struct structure) for the new process, and stores its address in the tsk local variable.

   c. Executes the alloc_thread_info macro to get a free memory area to store the thread_info structure and the Kernel Mode stack of the new process, and saves its address in the ti local variable. As explained in the earlier section "Identifying a Process," the size of this memory area is either 8 KB or 4 KB.

   d. Copies the contents of the current's process descriptor into the task_struct structure pointed to by tsk, then sets tsk->thread_info to ti.

   e. Copies the contents of the current's thread_info descriptor into the structure pointed to by ti, then sets ti->task to tsk.

   f. Sets the usage counter of the new process descriptor (tsk->usage) to 2 to specify that the process descriptor is in use and that the corresponding process is alive (its state is not EXIT_ZOMBIE or EXIT_DEAD).

   g. Returns the process descriptor pointer of the new process (tsk).

4. Checks whether the value stored in current->signal->rlim[RLIMIT_NPROC].rlim_cur is smaller than or equal to the current number of processes owned by the user. If so, an error code is returned, unless the process has root privileges. The

function gets the current number of processes owned by the user from a per-user data structure named `user_struct`. This data structure can be found through a pointer in the `user` field of the process descriptor.

5. Increases the usage counter of the `user_struct` structure (`tsk->user->__count` field) and the counter of the processes owned by the user (`tsk->user->processes`).

6. Checks that the number of processes in the system (stored in the `nr_threads` variable) does not exceed the value of the `max_threads` variable. The default value of this variable depends on the amount of RAM in the system. The general rule is that the space taken by all `thread_info` descriptors and Kernel Mode stacks cannot exceed 1/8 of the physical memory. However, the system administrator may change this value by writing in the */proc/sys/kernel/threads-max* file.

7. If the kernel functions implementing the execution domain and the executable format (see Chapter 20) of the new process are included in kernel modules, it increases their usage counters (see Appendix B).

8. Sets a few crucial fields related to the process state:

    a. Initializes the big kernel lock counter `tsk->lock_depth` to -1 (see the section "The Big Kernel Lock" in Chapter 5).

    b. Initializes the `tsk->did_exec` field to 0: it counts the number of `execve()` system calls issued by the process.

    c. Updates some of the flags included in the `tsk->flags` field that have been copied from the parent process: first clears the `PF_SUPERPRIV` flag, which indicates whether the process has used any of its superuser privileges, then sets the `PF_FORKNOEXEC` flag, which indicates that the child has not yet issued an `execve()` system call.

9. Stores the PID of the new process in the `tsk->pid` field.

10. If the `CLONE_PARENT_SETTID` flag in the `clone_flags` parameter is set, it copies the child's PID into the User Mode variable addressed by the `parent_tidptr` parameter.

11. Initializes the `list_head` data structures and the spin locks included in the child's process descriptor, and sets up several other fields related to pending signals, timers, and time statistics.

12. Invokes `copy_semundo()`, `copy_files()`, `copy_fs()`, `copy_sighand()`, `copy_signal()`, `copy_mm()`, and `copy_namespace()` to create new data structures and copy into them the values of the corresponding parent process data structures, unless specified differently by the `clone_flags` parameter.

13. Invokes `copy_thread()` to initialize the Kernel Mode stack of the child process with the values contained in the CPU registers when the `clone()` system call was issued (these values have been saved in the Kernel Mode stack of the parent, as described in Chapter 10). However, the function forces the value 0 into the field

corresponding to the eax register (this is the child's return value of the fork() or clone() system call). The thread.esp field in the descriptor of the child process is initialized with the base address of the child's Kernel Mode stack, and the address of an assembly language function (ret_from_fork()) is stored in the thread.eip field. If the parent process makes use of an I/O Permission Bitmap, the child gets a copy of such bitmap. Finally, if the CLONE_SETTLS flag is set, the child gets the TLS segment specified by the User Mode data structure pointed to by the tls parameter of the clone() system call.*

14. If either CLONE_CHILD_SETTID or CLONE_CHILD_CLEARTID is set in the clone_flags parameter, it copies the value of the child_tidptr parameter in the tsk->set_chid_tid or tsk->clear_child_tid field, respectively. These flags specify that the value of the variable pointed to by child_tidptr in the User Mode address space of the child has to be changed, although the actual write operations will be done later.

15. Turns off the TIF_SYSCALL_TRACE flag in the thread_info structure of the child, so that the ret_from_fork() function will not notify the debugging process about the system call termination (see the section "Entering and Exiting a System Call" in Chapter 10). (The system call tracing of the child is not disabled, because it is controlled by the PTRACE_SYSCALL flag in tsk->ptrace.)

16. Initializes the tsk->exit_signal field with the signal number encoded in the low bits of the clone_flags parameter, unless the CLONE_THREAD flag is set, in which case initializes the field to -1. As we'll see in the section "Process Termination" later in this chapter, only the death of the last member of a thread group (usually, the thread group leader) causes a signal notifying the parent of the thread group leader.

17. Invokes sched_fork() to complete the initialization of the scheduler data structure of the new process. The function also sets the state of the new process to TASK_RUNNING and sets the preempt_count field of the thread_info structure to 1, thus disabling kernel preemption (see the section "Kernel Preemption" in Chapter 5). Moreover, in order to keep process scheduling fair, the function shares the remaining timeslice of the parent between the parent and the child (see "The scheduler_tick() Function" in Chapter 7).

18. Sets the cpu field in the thread_info structure of the new process to the number of the local CPU returned by smp_processor_id().

19. Initializes the fields that specify the parenthood relationships. In particular, if CLONE_PARENT or CLONE_THREAD are set, it initializes tsk->real_parent and tsk->

---

* A careful reader might wonder how copy_thread() gets the value of the tls parameter of clone(), because tls is not passed to do_fork() and nested functions. As we'll see in Chapter 10, the parameters of the system calls are usually passed to the kernel by copying their values into some CPU register; thus, these values are saved in the Kernel Mode stack together with the other registers. The copy_thread() function just looks at the address saved in the Kernel Mode stack location corresponding to the value of esi.

parent to the value in current->real_parent; the parent of the child thus appears as the parent of the current process. Otherwise, it sets the same fields to current.

20. If the child does not need to be traced (CLONE_PTRACE flag not set), it sets the tsk->ptrace field to 0. This field stores a few flags used when a process is being traced by another process. In such a way, even if the current process is being traced, the child will not.

21. Executes the SET_LINKS macro to insert the new process descriptor in the process list.

22. If the child must be traced (PT_PTRACED flag in the tsk->ptrace field set), it sets tsk->parent to current->parent and inserts the child into the trace list of the debugger.

23. Invokes attach_pid( ) to insert the PID of the new process descriptor in the pidhash[PIDTYPE_PID] hash table.

24. If the child is a thread group leader (flag CLONE_THREAD cleared):

    a. Initializes tsk->tgid to tsk->pid.

    b. Initializes tsk->group_leader to tsk.

    c. Invokes three times attach_pid( ) to insert the child in the PID hash tables of type PIDTYPE_TGID, PIDTYPE_PGID, and PIDTYPE_SID.

25. Otherwise, if the child belongs to the thread group of its parent (CLONE_THREAD flag set):

    a. Initializes tsk->tgid to tsk->current->tgid.

    b. Initializes tsk->group_leader to the value in current->group_leader.

    c. Invokes attach_pid( ) to insert the child in the PIDTYPE_TGID hash table (more specifically, in the per-PID list of the current->group_leader process).

26. A new process has now been added to the set of processes: increases the value of the nr_threads variable.

27. Increases the total_forks variable to keep track of the number of forked processes.

28. Terminates by returning the child's process descriptor pointer (tsk).

Let's go back to what happens after do_fork() terminates. Now we have a complete child process in the runnable state. But it isn't actually running. It is up to the scheduler to decide when to give the CPU to this child. At some future process switch, the schedule bestows this favor on the child process by loading a few CPU registers with the values of the thread field of the child's process descriptor. In particular, esp is loaded with thread.esp (that is, with the address of child's Kernel Mode stack), and eip is loaded with the address of ret_from_fork( ). This assembly language function invokes the schedule_tail( ) function (which in turn invokes the finish_task_switch( ) function to complete the process switch; see the section "The schedule( ) Function" in Chapter 7), reloads all other registers with the values stored in the

stack, and forces the CPU back to User Mode. The new process then starts its execution right at the end of the `fork( )`, `vfork( )`, or `clone( )` system call. The value returned by the system call is contained in `eax`: the value is 0 for the child and equal to the PID for the child's parent. To understand how this is done, look back at what `copy_thread( )` does on the `eax` register of the child's process (step 13 of `copy_process( )`).

The child process executes the same code as the parent, except that the fork returns a 0 (see step 13 of `copy_process( )`). The developer of the application can exploit this fact, in a manner familiar to Unix programmers, by inserting a conditional statement in the program based on the PID value that forces the child to behave differently from the parent process.

Kernel Threads

Traditional Unix systems delegate some critical tasks to intermittently running processes, including flushing disk caches, swapping out unused pages, servicing network connections, and so on. Indeed, it is not efficient to perform these tasks in strict linear fashion; both their functions and the end user processes get better response if they are scheduled in the background. Because some of the system processes run only in Kernel Mode, modern operating systems delegate their functions to *kernel threads*, which are not encumbered with the unnecessary User Mode context. In Linux, kernel threads differ from regular processes in the following ways:

- Kernel threads run only in Kernel Mode, while regular processes run alternatively in Kernel Mode and in User Mode.
- Because kernel threads run only in Kernel Mode, they use only linear addresses greater than `PAGE_OFFSET`. Regular processes, on the other hand, use all four gigabytes of linear addresses, in either User Mode or Kernel Mode.

Creating a kernel thread

The `kernel_thread( )` function creates a new kernel thread. It receives as parameters the address of the kernel function to be executed (`fn`), the argument to be passed to that function (`arg`), and a set of clone flags (`flags`). The function essentially invokes `do_fork( )` as follows:

```
do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, pregs, 0, NULL, NULL);
```

The `CLONE_VM` flag avoids the duplication of the page tables of the calling process: this duplication would be a waste of time and memory, because the new kernel thread will not access the User Mode address space anyway. The `CLONE_UNTRACED` flag ensures that no process will be able to trace the new kernel thread, even if the calling process is being traced.

The `pregs` parameter passed to `do_fork( )` corresponds to the address in the Kernel Mode stack where the `copy_thread( )` function will find the initial values of the CPU

registers for the new thread. The `kernel_thread()` function builds up this stack area so that:

- The ebx and edx registers will be set by `copy_thread()` to the values of the parameters fn and arg, respectively.

- The eip register will be set to the address of the following assembly language fragment:

```
movl %edx,%eax
pushl %edx
call *%ebx
pushl %eax
call do_exit
```

Therefore, the new kernel thread starts by executing the fn(arg) function. If this function terminates, the kernel thread executes the _exit() system call passing to it the return value of fn() (see the section "Destroying Processes" later in this chapter).

Process 0

The ancestor of all processes, called *process 0*, the *idle process,* or, for historical reasons, the *swapper process*, is a kernel thread created from scratch during the initialization phase of Linux (see Appendix A). This ancestor process uses the following statically allocated data structures (data structures for all other processes are dynamically allocated):

- A process descriptor stored in the `init_task` variable, which is initialized by the `INIT_TASK` macro.

- A `thread_info` descriptor and a Kernel Mode stack stored in the `init_thread_union` variable and initialized by the `INIT_THREAD_INFO` macro.

- The following tables, which the process descriptor points to:
  — `init_mm`
  — `init_fs`
  — `init_files`
  — `init_signals`
  — `init_sighand`

  The tables are initialized, respectively, by the following macros:
  — `INIT_MM`
  — `INIT_FS`
  — `INIT_FILES`
  — `INIT_SIGNALS`
  — `INIT_SIGHAND`

- The master kernel Page Global Directory stored in `swapper_pg_dir` (see the section "Kernel Page Tables" in Chapter 2).

The `start_kernel()` function initializes all the data structures needed by the kernel, enables interrupts, and creates another kernel thread, named *process 1* (more commonly referred to as the *init process*):

```
kernel_thread(init, NULL, CLONE_FS|CLONE_SIGHAND);
```

The newly created kernel thread has PID 1 and shares all per-process kernel data structures with process 0. When selected by the scheduler, the *init* process starts executing the `init()` function.

After having created the *init* process, process 0 executes the `cpu_idle()` function, which essentially consists of repeatedly executing the `hlt` assembly language instruction with the interrupts enabled (see Chapter 4). Process 0 is selected by the scheduler only when there are no other processes in the `TASK_RUNNING` state.

In multiprocessor systems there is a process 0 for each CPU. Right after the power-on, the BIOS of the computer starts a single CPU while disabling the others. The swapper process running on CPU 0 initializes the kernel data structures, then enables the other CPUs and creates the additional *swapper* processes by means of the `copy_process()` function passing to it the value 0 as the new PID. Moreover, the kernel sets the `cpu` field of the `thread_info` descriptor of each forked process to the proper CPU index.

Process 1

The kernel thread created by process 0 executes the `init()` function, which in turn completes the initialization of the kernel. Then `init()` invokes the `execve()` system call to load the executable program *init*. As a result, the *init* kernel thread becomes a regular process having its own per-process kernel data structure (see Chapter 20). The *init* process stays alive until the system is shut down, because it creates and monitors the activity of all processes that implement the outer layers of the operating system.

Other kernel threads

Linux uses many other kernel threads. Some of them are created in the initialization phase and run until shutdown; others are created "on demand," when the kernel must execute a task that is better performed in its own execution context.

A few examples of kernel threads (besides process 0 and process 1) are:

keventd (also called events)
    Executes the functions in the `keventd_wq` workqueue (see Chapter 4).

kapmd
    Handles the events related to the Advanced Power Management (APM).

kswapd

Reclaims memory, as described in the section "Periodic Reclaiming" in Chapter 17.

pdflush

Flushes "dirty" buffers to disk to reclaim memory, as described in the section "The pdflush Kernel Threads" in Chapter 15.

kblockd

Executes the functions in the `kblockd_workqueue` workqueue. Essentially, it periodically activates the block device drivers, as described in the section "Activating the Block Device Driver" in Chapter 14.

ksoftirqd

Runs the tasklets (see section "Softirqs and Tasklets" in Chapter 4); there is one of these kernel threads for each CPU in the system.

Destroying Processes

Most processes "die" in the sense that they terminate the execution of the code they were supposed to run. When this occurs, the kernel must be notified so that it can release the resources owned by the process; this includes memory, open files, and any other odds and ends that we will encounter in this book, such as semaphores.

The usual way for a process to terminate is to invoke the `exit()` library function, which releases the resources allocated by the C library, executes each function registered by the programmer, and ends up invoking a system call that evicts the process from the system. The `exit()` library function may be inserted by the programmer explicitly. Additionally, the C compiler always inserts an `exit()` function call right after the last statement of the `main()` function.

Alternatively, the kernel may force a whole thread group to die. This typically occurs when a process in the group has received a signal that it cannot handle or ignore (see Chapter 11) or when an unrecoverable CPU exception has been raised in Kernel Mode while the kernel was running on behalf of the process (see Chapter 4).

Process Termination

In Linux 2.6 there are two system calls that terminate a User Mode application:

- The `exit_group()` system call, which terminates a full thread group, that is, a whole multithreaded application. The main kernel function that implements this system call is called `do_group_exit()`. This is the system call that should be invoked by the `exit()` C library function.

- The `_exit()` system call, which terminates a single process, regardless of any other process in the thread group of the victim. The main kernel function that

implements this system call is called do_exit(). This is the system call invoked, for instance, by the pthread_exit() function of the LinuxThreads library.

The do_group_exit() function

The do_group_exit() function kills all processes belonging to the thread group of current. It receives as a parameter the process termination code, which is either a value specified in the exit_group() system call (normal termination) or an error code supplied by the kernel (abnormal termination). The function executes the following operations:

1. Checks whether the SIGNAL_GROUP_EXIT flag of the exiting process is not zero, which means that the kernel already started an exit procedure for this thread group. In this case, it considers as exit code the value stored in current->signal->group_exit_code, and jumps to step 4.

2. Otherwise, it sets the SIGNAL_GROUP_EXIT flag of the process and stores the termination code in the current->signal->group_exit_code field.

3. Invokes the zap_other_threads() function to kill the other processes in the thread group of current, if any. In order to do this, the function scans the per-PID list in the PIDTYPE_TGID hash table corresponding to current->tgid; for each process in the list different from current, it sends a SIGKILL signal to it (see Chapter 11). As a result, all such processes will eventually execute the do_exit() function, and thus they will be killed.

4. Invokes the do_exit() function passing to it the process termination code. As we'll see below, do_exit() kills the process and never returns.

The do_exit() function

All process terminations are handled by the do_exit() function, which removes most references to the terminating process from kernel data structures. The do_exit() function receives as a parameter the process termination code and essentially executes the following actions:

1. Sets the PF_EXITING flag in the flag field of the process descriptor to indicate that the process is being eliminated.

2. Removes, if necessary, the process descriptor from a dynamic timer queue via the del_timer_sync() function (see Chapter 6).

3. Detaches from the process descriptor the data structures related to paging, semaphores, filesystem, open file descriptors, namespaces, and I/O Permission Bitmap, respectively, with the exit_mm(), exit_sem(), __exit_files(), __exit_fs(), exit_namespace(), and exit_thread() functions. These functions also remove each of these data structures if no other processes are sharing them.

4. If the kernel functions implementing the execution domain and the executable format (see Chapter 20) of the process being killed are included in kernel modules, the function decreases their usage counters.

5. Sets the `exit_code` field of the process descriptor to the process termination code. This value is either the `_exit()` or `exit_group()` system call parameter (normal termination), or an error code supplied by the kernel (abnormal termination).

6. Invokes the `exit_notify()` function to perform the following operations:

   a. Updates the parenthood relationships of both the parent process and the child processes. All child processes created by the terminating process become children of another process in the same thread group, if any is running, or otherwise of the *init* process.

   b. Checks whether the `exit_signal` process descriptor field of the process being terminated is different from -1, and whether the process is the last member of its thread group (notice that these conditions always hold for any normal process; see step 16 in the description of `copy_process()` in the earlier section "The clone(), fork(), and vfork() System Calls"). In this case, the function sends a signal (usually `SIGCHLD`) to the parent of the process being terminated to notify the parent about a child's death.

   c. Otherwise, if the `exit_signal` field is equal to -1 or the thread group includes other processes, the function sends a `SIGCHLD` signal to the parent only if the process is being traced (in this case the parent is the debugger, which is thus informed of the death of the lightweight process).

   d. If the `exit_signal` process descriptor field is equal to -1 and the process is not being traced, it sets the `exit_state` field of the process descriptor to `EXIT_DEAD`, and invokes `release_task()` to reclaim the memory of the remaining process data structures and to decrease the usage counter of the process descriptor (see the following section). The usage counter becomes equal to 1 (see step 3f in the `copy_process()` function), so that the process descriptor itself is not released right away.

   e. Otherwise, if the `exit_signal` process descriptor field is not equal to -1 or the process is being traced, it sets the `exit_state` field to `EXIT_ZOMBIE`. We'll see what happens to zombie processes in the following section.

   f. Sets the `PF_DEAD` flag in the `flags` field of the process descriptor (see the section "The schedule() Function" in Chapter 7).

7. Invokes the `schedule()` function (see Chapter 7) to select a new process to run. Because a process in an `EXIT_ZOMBIE` state is ignored by the scheduler, the process stops executing right after the `switch_to` macro in `schedule()` is invoked. As we'll see in Chapter 7, the scheduler will check the `PF_DEAD` flag and will decrease

the usage counter in the descriptor of the zombie process being replaced to denote the fact that the process is no longer alive.

Process Removal

The Unix operating system allows a process to query the kernel to obtain the PID of its parent process or the execution state of any of its children. A process may, for instance, create a child process to perform a specific task and then invoke some `wait( )`-like library function to check whether the child has terminated. If the child has terminated, its termination code will tell the parent process if the task has been carried out successfully.

To comply with these design choices, Unix kernels are not allowed to discard data included in a process descriptor field right after the process terminates. They are allowed to do so only after the parent process has issued a `wait( )`-like system call that refers to the terminated process. This is why the `EXIT_ZOMBIE` state has been introduced: although the process is technically dead, its descriptor must be saved until the parent process is notified.

What happens if parent processes terminate before their children? In such a case, the system could be flooded with zombie processes whose process descriptors would stay forever in RAM. As mentioned earlier, this problem is solved by forcing all orphan processes to become children of the *init* process. In this way, the *init* process will destroy the zombies while checking for the termination of one of its legitimate children through a `wait( )`-like system call.

The `release_task( )` function detaches the last data structures from the descriptor of a zombie process; it is applied on a zombie process in two possible ways: by the `do_exit()` function if the parent is not interested in receiving signals from the child, or by the `wait4()` or `waitpid()` system calls after a signal has been sent to the parent. In the latter case, the function also will reclaim the memory used by the process descriptor, while in the former case the memory reclaiming will be done by the scheduler (see Chapter 7). This function executes the following steps:

1. Decreases the number of processes belonging to the user owner of the termi-nated process. This value is stored in the `user_struct` structure mentioned ear-lier in the chapter (see step 4 of `copy_process()`).

2. If the process is being traced, the function removes it from the debugger's `ptrace_children` list and assigns the process back to its original parent.

3. Invokes `__exit_signal()` to cancel any pending signal and to release the `signal_struct` descriptor of the process. If the descriptor is no longer used by other lightweight processes, the function also removes this data structure. Moreover, the function invokes `exit_itimers()` to detach any POSIX interval timer from the process.

4. Invokes `__exit_sighand()` to get rid of the signal handlers.

5. Invokes `__unhash_process()`, which in turn:

   a. Decreases by 1 the `nr_threads` variable.

   b. Invokes `detach_pid()` twice to remove the process descriptor from the pidhash hash tables of type `PIDTYPE_PID` and `PIDTYPE_TGID`.

   c. If the process is a thread group leader, invokes again `detach_pid()` twice to remove the process descriptor from the `PIDTYPE_PGID` and `PIDTYPE_SID` hash tables.

   d. Uses the `REMOVE_LINKS` macro to unlink the process descriptor from the process list.

6. If the process is not a thread group leader, the leader is a zombie, and the process is the last member of the thread group, the function sends a signal to the parent of the leader to notify it of the death of the process.

7. Invokes the `sched_exit()` function to adjust the timeslice of the parent process (this step logically complements step 17 in the description of `copy_process()`)

8. Invokes `put_task_struct()` to decrease the process descriptor's usage counter; if the counter becomes zero, the function drops any remaining reference to the process:

   a. Decreases the usage counter (`__count` field) of the `user_struct` data structure of the user that owns the process (see step 5 of `copy_process()`), and releases that data structure if the usage counter becomes zero.

   b. Releases the process descriptor and the memory area used to contain the `thread_info` descriptor and the Kernel Mode stack.

Interrupts and Exceptions

An *interrupt* is usually defined as an event that alters the sequence of instructions executed by a processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside the CPU chip.

Interrupts are often divided into *synchronous* and *asynchronous* interrupts:

- *Synchronous* interrupts are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction.

- *Asynchronous* interrupts are generated by other hardware devices at arbitrary times with respect to the CPU clock signals.

Intel microprocessor manuals designate synchronous and asynchronous interrupts as *exceptions* and *interrupts*, respectively. We'll adopt this classification, although we'll occasionally use the term "interrupt signal" to designate both types together (synchronous as well as asynchronous).

Interrupts are issued by interval timers and I/O devices; for instance, the arrival of a keystroke from a user sets off an interrupt.

Exceptions, on the other hand, are caused either by programming errors or by anomalous conditions that must be handled by the kernel. In the first case, the kernel handles the exception by delivering to the current process one of the signals familiar to every Unix programmer. In the second case, the kernel performs all the steps needed to recover from the anomalous condition, such as a Page Fault or a request—via an assembly language instruction such as `int` or `sysenter`—for a kernel service.

We start by describing in the next section the motivation for introducing such signals. We then show how the well-known IRQs (Interrupt ReQuests) issued by I/O devices give rise to interrupts, and we detail how 80×86 processors handle interrupts and exceptions at the hardware level. Then we illustrate, in the section "Initializing the Interrupt Descriptor Table," how Linux initializes all the data structures

required by the 80×86 interrupt architecture. The remaining three sections describe how Linux handles interrupt signals at the software level.

One word of caution before moving on: in this chapter, we cover only "classic" interrupts common to all PCs; we do not cover the nonstandard interrupts of some architectures.

The Role of Interrupt Signals

As the name suggests, interrupt signals provide a way to divert the processor to code outside the normal flow of control. When an interrupt signal arrives, the CPU must stop what it's currently doing and switch to a new activity; it does this by saving the current value of the program counter (i.e., the content of the eip and cs registers) in the Kernel Mode stack and by placing an address related to the interrupt type into the program counter.

There are some things in this chapter that will remind you of the context switch described in the previous chapter, carried out when a kernel substitutes one process for another. But there is a key difference between interrupt handling and process switching: the code executed by an interrupt or by an exception handler is not a process. Rather, it is a kernel control path that runs at the expense of the same process that was running when the interrupt occurred (see the later section "Nested Execution of Exception and Interrupt Handlers"). As a kernel control path, the interrupt handler is lighter than a process (it has less context and requires less time to set up or tear down).

Interrupt handling is one of the most sensitive tasks performed by the kernel, because it must satisfy the following constraints:

- Interrupts can come anytime, when the kernel may want to finish something else it was trying to do. The kernel's goal is therefore to get the interrupt out of the way as soon as possible and defer as much processing as it can. For instance, suppose a block of data has arrived on a network line. When the hardware interrupts the kernel, it could simply mark the presence of data, give the processor back to whatever was running before, and do the rest of the processing later (such as moving the data into a buffer where its recipient process can find it, and then restarting the process). The activities that the kernel needs to perform in response to an interrupt are thus divided into a critical urgent part that the kernel executes right away and a deferrable part that is left for later.

- Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs. This should be allowed as much as possible, because it keeps the I/O devices busy (see the later section "Nested Execution of Exception and Interrupt Handlers"). As a result, the interrupt handlers must be coded so that the corresponding kernel control paths can be executed in a nested manner. When the last kernel control path terminates, the

kernel must be able to resume execution of the interrupted process or switch to another process if the interrupt signal has caused a rescheduling activity.

- Although the kernel may accept a new interrupt signal while handling a previous one, some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible because, according to the previous requirement, the kernel, and particularly the interrupt handlers, should run most of the time with the interrupts enabled.

Interrupts and Exceptions

The Intel documentation classifies interrupts and exceptions as follows:

- Interrupts:

  Maskable interrupts
  > All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts. A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.

  Nonmaskable interrupts
  > Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts. Nonmaskable interrupts are always recognized by the CPU.

- Exceptions:

  Processor-detected exceptions
  > Generated when the CPU detects an anomalous condition while executing an instruction. These are further divided into three groups, depending on the value of the eip register that is saved on the Kernel Mode stack when the CPU control unit raises the exception.

  Faults
  > Can generally be corrected; once corrected, the program is allowed to restart with no loss of continuity. The saved value of eip is the address of the instruction that caused the fault, and hence that instruction can be resumed when the exception handler terminates. As we'll see in the section "Page Fault Exception Handler" in Chapter 9, resuming the same instruction is necessary whenever the handler is able to correct the anomalous condition that caused the exception.

  Traps
  > Reported immediately following the execution of the trapping instruction; after the kernel returns control to the program, it is allowed to continue its execution with no loss of continuity. The saved value of eip is the address of the instruction that should be executed after the one that caused the trap. A trap is triggered only when there is no need to

reexecute the instruction that terminated. The main use of traps is for debugging purposes. The role of the interrupt signal in this case is to notify the debugger that a specific instruction has been executed (for instance, a breakpoint has been reached within a program). Once the user has examined the data provided by the debugger, she may ask that execution of the debugged program resume, starting from the next instruction.

Aborts

A serious error occurred; the control unit is in trouble, and it may be unable to store in the `eip` register the precise location of the instruction causing the exception. Aborts are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables. The interrupt signal sent by the control unit is an emergency signal used to switch control to the corresponding abort exception handler. This handler has no choice but to force the affected process to terminate.

Programmed exceptions

Occur at the request of the programmer. They are triggered by `int` or `int3` instructions; the `into` (check for overflow) and `bound` (check on address bound) instructions also give rise to a programmed exception when the condition they are checking is not true. Programmed exceptions are handled by the control unit as traps; they are often called *software interrupts*. Such exceptions have two common uses: to implement system calls and to notify a debugger of a specific event (see Chapter 10).

Each interrupt or exception is identified by a number ranging from 0 to 255; Intel calls this 8-bit unsigned number a *vector*. The vectors of nonmaskable interrupts and exceptions are fixed, while those of maskable interrupts can be altered by programming the Interrupt Controller (see the next section).

IRQs and Interrupts

Each hardware device controller capable of issuing interrupt requests usually has a single output line designated as the Interrupt ReQuest (*IRQ*) line.* All existing IRQ lines are connected to the input pins of a hardware circuit called the *Programmable Interrupt Controller*, which performs the following actions:

1. Monitors the IRQ lines, checking for raised signals. If two or more IRQ lines are raised, selects the one having the lower pin number.

2. If a raised signal occurs on an IRQ line:

    a. Converts the raised signal received into a corresponding vector.

---

* More sophisticated devices use several IRQ lines. For instance, a PCI card can use up to four IRQ lines.

---

b. Stores the vector in an Interrupt Controller I/O port, thus allowing the CPU to read it via the data bus.

c. Sends a raised signal to the processor INTR pin—that is, issues an interrupt.

d. Waits until the CPU acknowledges the interrupt signal by writing into one of the *Programmable Interrupt Controllers* (*PIC*) I/O ports; when this occurs, clears the INTR line.

3. Goes back to step 1.

The IRQ lines are sequentially numbered starting from 0; therefore, the first IRQ line is usually denoted as IRQ0. Intel's default vector associated with IRQn is $n+32$. As mentioned before, the mapping between IRQs and vectors can be modified by issuing suitable I/O instructions to the Interrupt Controller ports.

Each IRQ line can be selectively disabled. Thus, the PIC can be programmed to disable IRQs. That is, the PIC can be told to stop issuing interrupts that refer to a given IRQ line, or to resume issuing them. Disabled interrupts are not lost; the PIC sends them to the CPU as soon as they are enabled again. This feature is used by most interrupt handlers, because it allows them to process IRQs of the same type serially.

Selective enabling/disabling of IRQs is not the same as global masking/unmasking of maskable interrupts. When the IF flag of the eflags register is clear, each maskable interrupt issued by the PIC is temporarily ignored by the CPU. The cli and sti assembly language instructions, respectively, clear and set that flag.

Traditional PICs are implemented by connecting "in cascade" two 8259A-style external chips. Each chip can handle up to eight different IRQ input lines. Because the INT output line of the slave PIC is connected to the IRQ2 pin of the master PIC, the number of available IRQ lines is limited to 15.

The Advanced Programmable Interrupt Controller (APIC)

The previous description refers to PICs designed for uniprocessor systems. If the system includes a single CPU, the output line of the master PIC can be connected in a straightforward way to the INTR pin the CPU. However, if the system includes two or more CPUs, this approach is no longer valid and more sophisticated PICs are needed.

Being able to deliver interrupts to each CPU in the system is crucial for fully exploiting the parallelism of the SMP architecture. For that reason, Intel introduced starting with Pentium III a new component designated as the *I/O Advanced Programmable Interrupt Controller* (*I/O APIC*). This chip is the advanced version of the old 8259A Programmable Interrupt Controller; to support old operating systems, recent motherboards include both types of chip. Moreover, all current 80×86 microprocessors include a *local APIC*. Each local APIC has 32-bit registers, an internal clock; a local timer device; and two additional IRQ lines, LINT0 and LINT1, reserved for *local*

*APIC interrupt*s. All local APICs are connected to an external I/O APIC, giving rise to a multi-APIC system.

Figure 4-1 illustrates in a schematic way the structure of a multi-APIC system. An *APIC bus* connects the "frontend" I/O APIC to the local APICs. The IRQ lines coming from the devices are connected to the I/O APIC, which therefore acts as a router with respect to the local APICs. In the motherboards of the Pentium III and earlier processors, the APIC bus was a serial three-line bus; starting with the Pentium 4, the APIC bus is implemented by means of the system bus. However, because the APIC bus and its messages are invisible to software, we won't give further details.
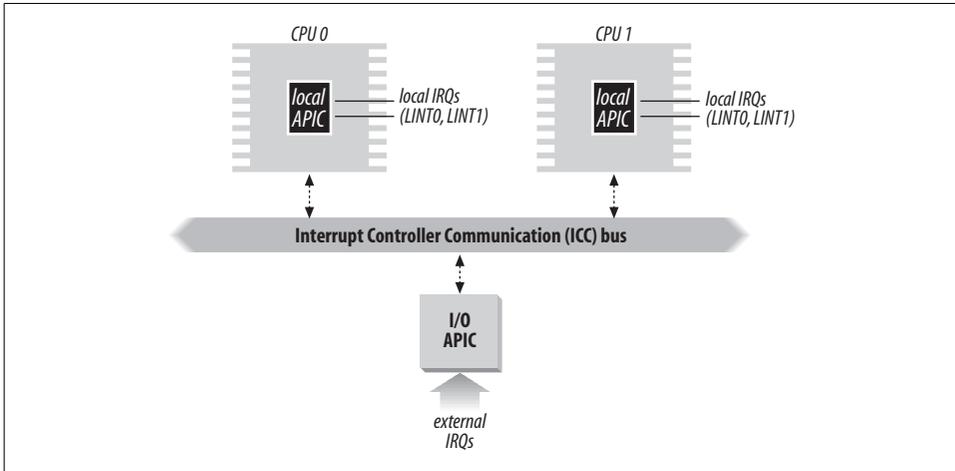
Figure 4-1. Multi-APIC system

The I/O APIC consists of a set of 24 IRQ lines, a 24-entry *Interrupt Redirection Table*, programmable registers, and a message unit for sending and receiving APIC messages over the APIC bus. Unlike IRQ pins of the 8259A, interrupt priority is not related to pin number: each entry in the Redirection Table can be individually programmed to indicate the interrupt vector and priority, the destination processor, and how the processor is selected. The information in the Redirection Table is used to translate each external IRQ signal into a message to one or more local APIC units via the APIC bus.

Interrupt requests coming from external hardware devices can be distributed among the available CPUs in two ways:

Static distribution

The IRQ signal is delivered to the local APICs listed in the corresponding Redirection Table entry. The interrupt is delivered to one specific CPU, to a subset of CPUs, or to all CPUs at once (broadcast mode).

Dynamic distribution

> The IRQ signal is delivered to the local APIC of the processor that is executing the process with the lowest priority.

> Every local APIC has a programmable *task priority register* (TPR), which is used to compute the priority of the currently running process. Intel expects this register to be modified in an operating system kernel by each process switch.

> If two or more CPUs share the lowest priority, the load is distributed between them using a technique called *arbitration*. Each CPU is assigned a different arbitration priority ranging from 0 (lowest) to 15 (highest) in the arbitration priority register of the local APIC.

> Every time an interrupt is delivered to a CPU, its corresponding arbitration priority is automatically set to 0, while the arbitration priority of any other CPU is increased. When the arbitration priority register becomes greater than 15, it is set to the previous arbitration priority of the winning CPU increased by 1. Therefore, interrupts are distributed in a round-robin fashion among CPUs with the same task priority.[*]

Besides distributing interrupts among processors, the multi-APIC system allows CPUs to generate *interprocessor interrupts*. When a CPU wishes to send an interrupt to another CPU, it stores the interrupt vector and the identifier of the target's local APIC in the Interrupt Command Register (ICR) of its own local APIC. A message is then sent via the APIC bus to the target's local APIC, which therefore issues a corresponding interrupt to its own CPU.

Interprocessor interrupts (in short, IPIs) are a crucial component of the SMP architecture. They are actively used by Linux to exchange messages among CPUs (see later in this chapter).

Many of the current uniprocessor systems include an I/O APIC chip, which may be configured in two distinct ways:

- As a standard 8259A-style external PIC connected to the CPU. The local APIC is disabled and the two LINT0 and LINT1 local IRQ lines are configured, respectively, as the INTR and NMI pins.

- As a standard external I/O APIC. The local APIC is enabled, and all external interrupts are received through the I/O APIC.

---

[*] The Pentium 4 local APIC doesn't have an arbitration priority register; the arbitration mechanism is hidden in the bus arbitration circuitry. The Intel manuals state that if the operating system kernel does not regularly update the task priority registers, performance may be suboptimal because interrupts might always be serviced by the same CPU.

Exceptions

The 80×86 microprocessors issue roughly 20 different exceptions.[*] The kernel must provide a dedicated exception handler for each exception type. For some exceptions, the CPU control unit also generates a *hardware error code* and pushes it on the Kernel Mode stack before starting the exception handler.

The following list gives the vector, the name, the type, and a brief description of the exceptions found in 80×86 processors. Additional information may be found in the Intel technical documentation.

0 - "Divide error" (fault)
> Raised when a program issues an integer division by 0.

1- "Debug" (trap or fault)
> Raised when the TF flag of eflags is set (quite useful to implement *single-step execution* of a debugged program) or when the address of an instruction or operand falls within the range of an active debug register (see the section "Hardware Context" in Chapter 3).

2 - Not used
> Reserved for nonmaskable interrupts (those that use the NMI pin).

3 - "Breakpoint" (trap)
> Caused by an int3 (breakpoint) instruction (usually inserted by a debugger).

4 - "Overflow" (trap)
> An into (check for overflow) instruction has been executed while the OF (overflow) flag of eflags is set.

5 - "Bounds check" (fault)
> A bound (check on address bound) instruction is executed with the operand outside of the valid address bounds.

6 - "Invalid opcode" (fault)
> The CPU execution unit has detected an invalid opcode (the part of the machine instruction that determines the operation performed).

7 - "Device not available" (fault)
> An ESCAPE, MMX, or SSE/SSE2 instruction has been executed with the TS flag of cr0 set (see the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3).

8 - "Double fault" (abort)
> Normally, when the CPU detects an exception while trying to call the handler for a prior exception, the two exceptions can be handled serially. In a few cases, however, the processor cannot handle them serially, so it raises this exception.

---

[*] The exact number depends on the processor model.

---

9 - "Coprocessor segment overrun" (abort)
Problems with the external mathematical coprocessor (applies only to old 80386 microprocessors).

10 - "Invalid TSS" (fault)
The CPU has attempted a context switch to a process having an invalid Task State Segment.

11 - "Segment not present" (fault)
A reference was made to a segment not present in memory (one in which the `Segment-Present` flag of the Segment Descriptor was cleared).

12 - "Stack segment fault" (fault)
The instruction attempted to exceed the stack segment limit, or the segment identified by `ss` is not present in memory.

13 - "General protection" (fault)
One of the protection rules in the protected mode of the 80×86 has been violated.

14 - "Page Fault" (fault)
The addressed page is not present in memory, the corresponding Page Table entry is null, or a violation of the paging protection mechanism has occurred.

15 - Reserved by Intel

16 - "Floating-point error" (fault)
The floating-point unit integrated into the CPU chip has signaled an error condition, such as numeric overflow or division by 0.[*]

17 - "Alignment check" (fault)
The address of an operand is not correctly aligned (for instance, the address of a long integer is not a multiple of 4).

18 - "Machine check" (abort)
A machine-check mechanism has detected a CPU or bus error.

19 - "SIMD floating point exception" (fault)
The SSE or SSE2 unit integrated in the CPU chip has signaled an error condition on a floating-point operation.

The values from 20 to 31 are reserved by Intel for future development. As illustrated in Table 4-1, each exception is handled by a specific exception handler (see the section "Exception Handling" later in this chapter), which usually sends a Unix signal to the process that caused the exception.

---

[*] The 80×86 microprocessors also generate this exception when performing a signed division whose result cannot be stored as a signed integer (for instance, a division between –2,147,483,648 and –1).

Table 4-1. Signals sent by the exception handlers

#	Exception	Exception handler	Signal
0	Divide error	`divide_error()`	SIGFPE
1	Debug	`debug()`	SIGTRAP
2	NMI	`nmi()`	None
3	Breakpoint	`int3()`	SIGTRAP
4	Overflow	`overflow()`	SIGSEGV
5	Bounds check	`bounds()`	SIGSEGV
6	Invalid opcode	`invalid_op()`	SIGILL
7	Device not available	`device_not_available()`	None
8	Double fault	`doublefault_fn()`	None
9	Coprocessor segment overrun	`coprocessor_segment_overrun()`	SIGFPE
10	Invalid TSS	`invalid_TSS()`	SIGSEGV
11	Segment not present	`segment_not_present()`	SIGBUS
12	Stack segment fault	`stack_segment()`	SIGBUS
13	General protection	`general_protection()`	SIGSEGV
14	Page Fault	`page_fault()`	SIGSEGV
15	Intel-reserved	None	None
16	Floating-point error	`coprocessor_error()`	SIGFPE
17	Alignment check	`alignment_check()`	SIGBUS
18	Machine check	`machine_check()`	None
19	SIMD floating point	`simd_coprocessor_error()`	SIGFPE

Interrupt Descriptor Table

A system table called *Interrupt Descriptor Table* (IDT) associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler. The IDT must be properly initialized before the kernel enables interrupts.

The IDT format is similar to that of the GDT and the LDTs examined in Chapter 2. Each entry corresponds to an interrupt or an exception vector and consists of an 8-byte descriptor. Thus, a maximum of $256 \times 8 = 2048$ bytes are required to store the IDT.

The idtr CPU register allows the IDT to be located anywhere in memory: it specifies both the IDT base linear address and its limit (maximum length). It must be initialized before enabling interrupts by using the lidt assembly language instruction.

The IDT may include three types of descriptors; Figure 4-2 illustrates the meaning of the 64 bits included in each of them. In particular, the value of the Type field encoded in the bits 40–43 identifies the descriptor type.
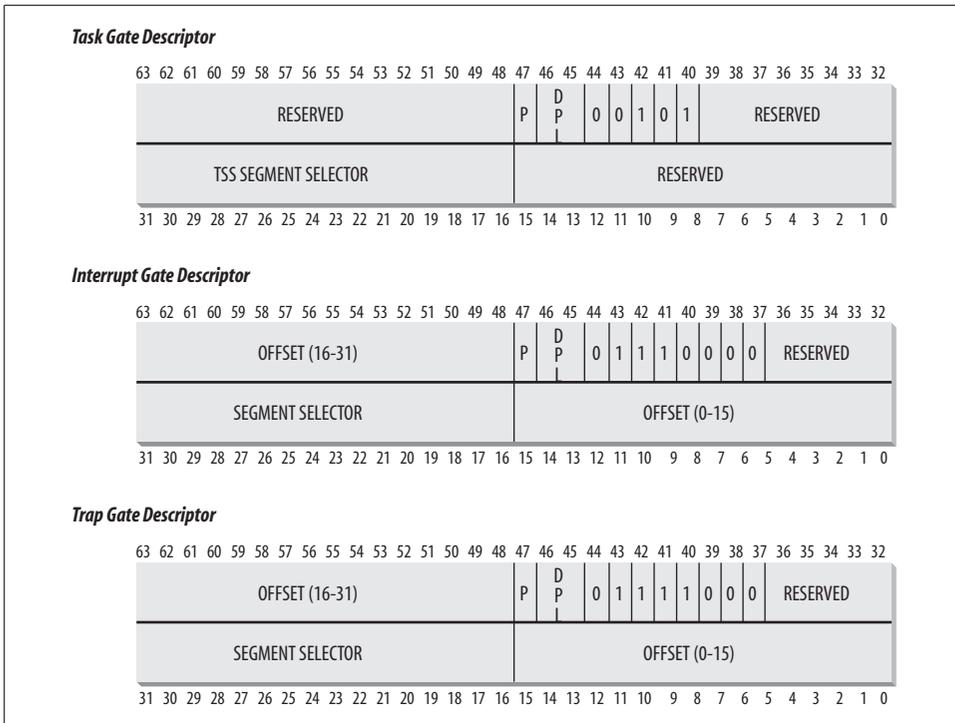
Figure 4-2. Gate descriptors' format

The descriptors are:

Task gate
> Includes the TSS selector of the process that must replace the current one when an interrupt signal occurs.

Interrupt gate
> Includes the Segment Selector and the offset inside the segment of an interrupt or exception handler. While transferring control to the proper segment, the processor clears the IF flag, thus disabling further maskable interrupts.

Trap gate
> Similar to an interrupt gate, except that while transferring control to the proper segment, the processor does not modify the IF flag.

As we'll see in the later section "Interrupt, Trap, and System Gates," Linux uses interrupt gates to handle interrupts and trap gates to handle exceptions.*

---

* The "Double fault" exception, which denotes a type of kernel misbehavior, is the only exception handled by means of a task gate (see the section "Exception Handling" later in this chapter.).

Hardware Handling of Interrupts and Exceptions

We now describe how the CPU control unit handles interrupts and exceptions. We assume that the kernel has been initialized, and thus the CPU is operating in Protected Mode.

After executing an instruction, the `cs` and `eip` pair of registers contain the logical address of the next instruction to be executed. Before dealing with that instruction, the control unit checks whether an interrupt or an exception occurred while the control unit executed the previous instruction. If one occurred, the control unit does the following:

1. Determines the vector i ($0 \le i \le 255$) associated with the interrupt or the exception.

2. Reads the ith entry of the IDT referred by the `idtr` register (we assume in the following description that the entry contains an interrupt or a trap gate).

3. Gets the base address of the GDT from the `gdtr` register and looks in the GDT to read the Segment Descriptor identified by the selector in the IDT entry. This descriptor specifies the base address of the segment that includes the interrupt or exception handler.

4. Makes sure the interrupt was issued by an authorized source. First, it compares the Current Privilege Level (CPL), which is stored in the two least significant bits of the `cs` register, with the Descriptor Privilege Level (DPL) of the Segment Descriptor included in the GDT. Raises a "General protection" exception if the CPL is lower than the DPL, because the interrupt handler cannot have a lower privilege than the program that caused the interrupt. For programmed exceptions, makes a further security check: compares the CPL with the DPL of the gate descriptor included in the IDT and raises a "General protection" exception if the DPL is lower than the CPL. This last check makes it possible to prevent access by user applications to specific trap or interrupt gates.

5. Checks whether a change of privilege level is taking place—that is, if CPL is different from the selected Segment Descriptor's DPL. If so, the control unit must start using the stack that is associated with the new privilege level. It does this by performing the following steps:

   a. Reads the `tr` register to access the TSS segment of the running process.

   b. Loads the `ss` and `esp` registers with the proper values for the stack segment and stack pointer associated with the new privilege level. These values are found in the TSS (see the section "Task State Segment" in Chapter 3).

   c. In the new stack, it saves the previous values of `ss` and `esp`, which define the logical address of the stack associated with the old privilege level.

6. If a fault has occurred, it loads `cs` and `eip` with the logical address of the instruction that caused the exception so that it can be executed again.

7. Saves the contents of `eflags`, `cs`, and `eip` in the stack.

8. If the exception carries a hardware error code, it saves it on the stack.

9. Loads `cs` and `eip`, respectively, with the Segment Selector and the Offset fields of the Gate Descriptor stored in the ith entry of the IDT. These values define the logical address of the first instruction of the interrupt or exception handler.

The last step performed by the control unit is equivalent to a jump to the interrupt or exception handler. In other words, the instruction processed by the control unit after dealing with the interrupt signal is the first instruction of the selected handler.

After the interrupt or exception is processed, the corresponding handler must relinquish control to the interrupted process by issuing the `iret` instruction, which forces the control unit to:

1. Load the `cs`, `eip`, and `eflags` registers with the values saved on the stack. If a hardware error code has been pushed in the stack on top of the `eip` contents, it must be popped before executing `iret`.

2. Check whether the CPL of the handler is equal to the value contained in the two least significant bits of `cs` (this means the interrupted process was running at the same privilege level as the handler). If so, `iret` concludes execution; otherwise, go to the next step.

3. Load the `ss` and `esp` registers from the stack and return to the stack associated with the old privilege level.

4. Examine the contents of the `ds`, `es`, `fs`, and `gs` segment registers; if any of them contains a selector that refers to a Segment Descriptor whose DPL value is lower than CPL, clear the corresponding segment register. The control unit does this to forbid User Mode programs that run with a CPL equal to 3 from using segment registers previously used by kernel routines (with a DPL equal to 0). If these registers were not cleared, malicious User Mode programs could exploit them in order to access the kernel address space.

Nested Execution of Exception and Interrupt Handlers

Every interrupt or exception gives rise to a *kernel control path* or separate sequence of instructions that execute in Kernel Mode on behalf of the current process. For instance, when an I/O device raises an interrupt, the first instructions of the corresponding kernel control path are those that save the contents of the CPU registers in the Kernel Mode stack, while the last are those that restore the contents of the registers.

Kernel control paths may be arbitrarily nested; an interrupt handler may be interrupted by another interrupt handler, thus giving rise to a nested execution of kernel control paths, as shown in Figure 4-3. As a result, the last instructions of a kernel control path that is taking care of an interrupt do not always put the current process back into User Mode: if the level of nesting is greater than 1, these instructions will put into execution the kernel control path that was interrupted last, and the CPU will continue to run in Kernel Mode.
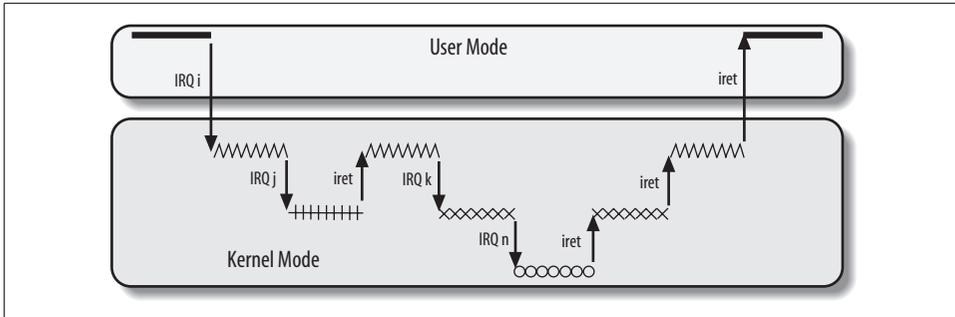
Figure 4-3. An example of nested execution of kernel control paths

The price to pay for allowing nested kernel control paths is that an interrupt handler must never block, that is, no process switch can take place until an interrupt handler is running. In fact, all the data needed to resume a nested kernel control path is stored in the Kernel Mode stack, which is tightly bound to the current process.

Assuming that the kernel is bug free, most exceptions can occur only while the CPU is in User Mode. Indeed, they are either caused by programming errors or triggered by debuggers. However, the "Page Fault" exception may occur in Kernel Mode. This happens when the process attempts to address a page that belongs to its address space but is not currently in RAM. While handling such an exception, the kernel may suspend the current process and replace it with another one until the requested page is available. The kernel control path that handles the "Page Fault" exception resumes execution as soon as the process gets the processor again.

Because the "Page Fault" exception handler never gives rise to further exceptions, at most two kernel control paths associated with exceptions (the first one caused by a system call invocation, the second one caused by a Page Fault) may be stacked, one on top of the other.

In contrast to exceptions, interrupts issued by I/O devices do not refer to data structures specific to the current process, although the kernel control paths that handle them run on behalf of that process. As a matter of fact, it is impossible to predict which process will be running when a given interrupt occurs.

An interrupt handler may preempt both other interrupt handlers and exception handlers. Conversely, an exception handler never preempts an interrupt handler. The only exception that can be triggered in Kernel Mode is "Page Fault," which we just described. But interrupt handlers never perform operations that can induce page faults, and thus, potentially, a process switch.

Linux interleaves kernel control paths for two major reasons:

- To improve the throughput of programmable interrupt controllers and device controllers. Assume that a device controller issues a signal on an IRQ line: the PIC transforms it into an external interrupt, and then both the PIC and the

device controller remain blocked until the PIC receives an acknowledgment from the CPU. Thanks to kernel control path interleaving, the kernel is able to send the acknowledgment even when it is handling a previous interrupt.

- To implement an interrupt model without priority levels. Because each interrupt handler may be deferred by another one, there is no need to establish predefined priorities among hardware devices. This simplifies the kernel code and improves its portability.

On multiprocessor systems, several kernel control paths may execute concurrently. Moreover, a kernel control path associated with an exception may start executing on a CPU and, due to a process switch, migrate to another CPU.

Initializing the Interrupt Descriptor Table

Now that we understand what the 80×86 microprocessors do with interrupts and exceptions at the hardware level, we can move on to describe how the Interrupt Descriptor Table is initialized.

Remember that before the kernel enables the interrupts, it must load the initial address of the IDT table into the `idtr` register and initialize all the entries of that table. This activity is done while initializing the system (see Appendix A).

The `int` instruction allows a User Mode process to issue an interrupt signal that has an arbitrary vector ranging from 0 to 255. Therefore, initialization of the IDT must be done carefully, to block illegal interrupts and exceptions simulated by User Mode processes via `int` instructions. This can be achieved by setting the DPL field of the particular Interrupt or Trap Gate Descriptor to 0. If the process attempts to issue one of these interrupt signals, the control unit checks the CPL value against the DPL field and issues a "General protection" exception.

In a few cases, however, a User Mode process must be able to issue a programmed exception. To allow this, it is sufficient to set the DPL field of the corresponding Interrupt or Trap Gate Descriptors to 3—that is, as high as possible.

Let's now see how Linux implements this strategy.

Interrupt, Trap, and System Gates

As mentioned in the earlier section "Interrupt Descriptor Table," Intel provides three types of interrupt descriptors: Task, Interrupt, and Trap Gate Descriptors. Linux uses a slightly different breakdown and terminology from Intel when classifying the interrupt descriptors included in the Interrupt Descriptor Table:

Interrupt gate

An Intel interrupt gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). All Linux interrupt handlers are activated by means of interrupt gates, and all are restricted to Kernel Mode.

System gate

An Intel trap gate that can be accessed by a User Mode process (the gate's DPL field is equal to 3). The three Linux exception handlers associated with the vectors 4, 5, and 128 are activated by means of system gates, so the three assembly language instructions `into`, `bound`, and `int $0x80` can be issued in User Mode.

System interrupt gate

An Intel interrupt gate that can be accessed by a User Mode process (the gate's DPL field is equal to 3). The exception handler associated with the vector 3 is activated by means of a system interrupt gate, so the assembly language instruction `int3` can be issued in User Mode.

Trap gate

An Intel trap gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). Most Linux exception handlers are activated by means of trap gates.

Task gate

An Intel task gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). The Linux handler for the "Double fault" exception is activated by means of a task gate.

The following architecture-dependent functions are used to insert gates in the IDT:

`set_intr_gate(n,addr)`

Inserts an interrupt gate in the nth IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to `addr`, which is the address of the interrupt handler. The DPL field is set to 0.

`set_system_gate(n,addr)`

Inserts a trap gate in the nth IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to `addr`, which is the address of the exception handler. The DPL field is set to 3.

`set_system_intr_gate(n,addr)`

Inserts an interrupt gate in the nth IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to `addr`, which is the address of the exception handler. The DPL field is set to 3.

`set_trap_gate(n,addr)`

Similar to the previous function, except the DPL field is set to 0.

```
set_task_gate(n,gdt)
```
Inserts a task gate in the *n*th IDT entry. The Segment Selector inside the gate stores the index in the GDT of the TSS containing the function to be activated. The Offset field is set to 0, while the DPL field is set to 3.

Preliminary Initialization of the IDT

The IDT is initialized and used by the BIOS routines while the computer still operates in Real Mode. Once Linux takes over, however, the IDT is moved to another area of RAM and initialized a second time, because Linux does not use any BIOS routine (see Appendix A).

The IDT is stored in the idt_table table, which includes 256 entries. The 6-byte idt_descr variable stores both the size of the IDT and its address and is used in the system initialization phase when the kernel sets up the idtr register with the lidt assembly language instruction.[*]

During kernel initialization, the setup_idt( ) assembly language function starts by filling all 256 entries of idt_table with the same interrupt gate, which refers to the ignore_int( ) interrupt handler:

```
setup_idt:
    lea ignore_int, %edx
    movl $(__KERNEL_CS << 16), %eax
    movw %dx, %ax       /* selector = 0x0010 = cs */
    movw $0x8e00, %dx   /* interrupt gate, dpl=0, present */
    lea idt_table, %edi
    mov $256, %ecx
rp_sidt:
    movl %eax, (%edi)
    movl %edx, 4(%edi)
    addl $8, %edi
    dec %ecx
    jne rp_sidt
    ret
```

The ignore_int( ) interrupt handler, which is in assembly language, may be viewed as a null handler that executes the following actions:

1. Saves the content of some registers in the stack.
2. Invokes the printk( ) function to print an "Unknown interrupt" system message.
3. Restores the register contents from the stack.
4. Executes an iret instruction to restart the interrupted program.

---

[*] Some old Pentium models have the notorious "f00f" bug, which allows User Mode programs to freeze the system. When executing on such CPUs, Linux uses a workaround based on initializing the idtr register with a fix-mapped read-only linear address pointing to the actual IDT (see the section "Fix-Mapped Linear Addresses" in Chapter 2).

The `ignore_int( )` handler should never be executed. The occurrence of "Unknown interrupt" messages on the console or in the log files denotes either a hardware problem (an I/O device is issuing unforeseen interrupts) or a kernel problem (an interrupt or exception is not being handled properly).

Following this preliminary initialization, the kernel makes a second pass in the IDT to replace some of the null handlers with meaningful trap and interrupt handlers. Once this is done, the IDT includes a specialized interrupt, trap, or system gate for each different exception issued by the control unit and for each IRQ recognized by the interrupt controller.

The next two sections illustrate in detail how this is done for exceptions and interrupts.

Exception Handling

Most exceptions issued by the CPU are interpreted by Linux as error conditions. When one of them occurs, the kernel sends a signal to the process that caused the exception to notify it of an anomalous condition. If, for instance, a process performs a division by zero, the CPU raises a "Divide error" exception, and the corresponding exception handler sends a `SIGFPE` signal to the current process, which then takes the necessary steps to recover or (if no signal handler is set for that signal) abort.

There are a couple of cases, however, where Linux exploits CPU exceptions to manage hardware resources more efficiently. A first case is already described in the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3. The "Device not available" exception is used together with the `TS` flag of the `cr0` register to force the kernel to load the floating point registers of the CPU with new values. A second case involves the "Page Fault" exception, which is used to defer allocating new page frames to the process until the last possible moment. The corresponding handler is complex because the exception may, or may not, denote an error condition (see the section "Page Fault Exception Handler" in Chapter 9).

Exception handlers have a standard structure consisting of three steps:

1. Save the contents of most registers in the Kernel Mode stack (this part is coded in assembly language).
2. Handle the exception by means of a high-level C function.
3. Exit from the handler by means of the `ret_from_exception( )` function.

To take advantage of exceptions, the IDT must be properly initialized with an exception handler function for each recognized exception. It is the job of the `trap_init( )` function to insert the final values—the functions that handle the exceptions—into all IDT entries that refer to nonmaskable interrupts and exceptions. This is accomplished through the `set_trap_gate()`, `set_intr_gate()`, `set_system_gate()`, `set_system_intr_gate()`, and `set_task_gate()` functions:

```
set_trap_gate(0,&divide_error);
set_trap_gate(1,&debug);
```

```
    set_intr_gate(2,&nmi);
    set_system_intr_gate(3,&int3);
    set_system_gate(4,&overflow);
    set_system_gate(5,&bounds);
    set_trap_gate(6,&invalid_op);
    set_trap_gate(7,&device_not_available);
    set_task_gate(8,31);
    set_trap_gate(9,&coprocessor_segment_overrun);
    set_trap_gate(10,&invalid_TSS);
    set_trap_gate(11,&segment_not_present);
    set_trap_gate(12,&stack_segment);
    set_trap_gate(13,&general_protection);
    set_intr_gate(14,&page_fault);
    set_trap_gate(16,&coprocessor_error);
    set_trap_gate(17,&alignment_check);
    set_trap_gate(18,&machine_check);
    set_trap_gate(19,&simd_coprocessor_error);
    set_system_gate(128,&system_call);
```

The "Double fault" exception is handled by means of a task gate instead of a trap or system gate, because it denotes a serious kernel misbehavior. Thus, the exception handler that tries to print out the register values does not trust the current value of the esp register. When such an exception occurs, the CPU fetches the Task Gate Descriptor stored in the entry at index 8 of the IDT. This descriptor points to the special TSS segment descriptor stored in the 32nd entry of the GDT. Next, the CPU loads the eip and esp registers with the values stored in the corresponding TSS segment. As a result, the processor executes the doublefault_fn() exception handler on its own private stack.

Now we will look at what a typical exception handler does once it is invoked. Our description of exception handling will be a bit sketchy for lack of space. In particular we won't be able to cover:

1. The signal codes (see Table 11-8 in Chapter 11) sent by some handlers to the User Mode processes.

2. Exceptions that occur when the kernel is operating in MS-DOS emulation mode (vm86 mode), which must be dealt with differently.

3. "Debug" exceptions.

Saving the Registers for the Exception Handler

Let's use handler_name to denote the name of a generic exception handler. (The actual names of all the exception handlers appear on the list of macros in the previous section.) Each exception handler starts with the following assembly language instructions:

```
handler_name:
    pushl $0 /* only for some exceptions */
    pushl $do_handler_name
    jmp error_code
```

If the control unit is not supposed to automatically insert a hardware error code on the stack when the exception occurs, the corresponding assembly language fragment includes a `pushl $0` instruction to pad the stack with a null value. Then the address of the high-level C function is pushed on the stack; its name consists of the exception handler name prefixed by `do_`.

The assembly language fragment labeled as `error_code` is the same for all exception handlers except the one for the "Device not available" exception (see the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3). The code performs the following steps:

1. Saves the registers that might be used by the high-level C function on the stack.

2. Issues a `cld` instruction to clear the direction flag `DF` of `eflags`, thus making sure that autoincreases on the `edi` and `esi` registers will be used with string instructions.[*]

3. Copies the hardware error code saved in the stack at location `esp+36` in `edx`. Stores the value –1 in the same stack location. As we'll see in the section "Reexecution of System Calls" in Chapter 11, this value is used to separate `0x80` exceptions from other exceptions.

4. Loads `edi` with the address of the high-level `do_handler_name()` C function saved in the stack at location `esp+32`; writes the contents of `es` in that stack location.

5. Loads in the `eax` register the current top location of the Kernel Mode stack. This address identifies the memory cell containing the last register value saved in step 1.

6. Loads the user data Segment Selector into the `ds` and `es` registers.

7. Invokes the high-level C function whose address is now stored in `edi`.

The invoked function receives its arguments from the `eax` and `edx` registers rather than from the stack. We have already run into a function that gets its arguments from the CPU registers: the `__switch_to()` function, discussed in the section "Performing the Process Switch" in Chapter 3.

Entering and Leaving the Exception Handler

As already explained, the names of the C functions that implement exception handlers always consist of the prefix `do_` followed by the handler name. Most of these functions invoke the `do_trap()` function to store the hardware error code and the

---

[*] A single assembly language "string instruction," such as `rep;movsb`, is able to act on a whole block of data (string).

exception vector in the process descriptor of current, and then send a suitable signal to that process:

```
current->thread.error_code = error_code;
current->thread.trap_no = vector;
force_sig(sig_number, current);
```

The current process takes care of the signal right after the termination of the exception handler. The signal will be handled either in User Mode by the process's own signal handler (if it exists) or in Kernel Mode. In the latter case, the kernel usually kills the process (see Chapter 11). The signals sent by the exception handlers are listed in Table 4-1.

The exception handler always checks whether the exception occurred in User Mode or in Kernel Mode and, in the latter case, whether it was due to an invalid argument passed to a system call. We'll describe in the section "Dynamic Address Checking: The Fix-up Code" in Chapter 10 how the kernel defends itself against invalid arguments passed to system calls. Any other exception raised in Kernel Mode is due to a kernel bug. In this case, the exception handler knows the kernel is misbehaving. In order to avoid data corruption on the hard disks, the handler invokes the die( ) function, which prints the contents of all CPU registers on the console (this dump is called *kernel oops*) and terminates the current process by calling do_exit( ) (see "Process Termination" in Chapter 3).

When the C function that implements the exception handling terminates, the code performs a jmp instruction to the ret_from_exception( ) function. This function is described in the later section "Returning from Interrupts and Exceptions."

Interrupt Handling

As we explained earlier, most exceptions are handled simply by sending a Unix signal to the process that caused the exception. The action to be taken is thus deferred until the process receives the signal; as a result, the kernel is able to process the exception quickly.

This approach does not hold for interrupts, because they frequently arrive long after the process to which they are related (for instance, a process that requested a data transfer) has been suspended and a completely unrelated process is running. So it would make no sense to send a Unix signal to the current process.

Interrupt handling depends on the type of interrupt. For our purposes, we'll distinguish three main classes of interrupts:

I/O interrupts
> An I/O device requires attention; the corresponding interrupt handler must query the device to determine the proper course of action. We cover this type of interrupt in the later section "I/O Interrupt Handling."

Timer interrupts

Some timer, either a local APIC timer or an external timer, has issued an inter-rupt; this kind of interrupt tells the kernel that a fixed-time interval has elapsed. These interrupts are handled mostly as I/O interrupts; we discuss the peculiar characteristics of timer interrupts in Chapter 6.

Interprocessor interrupts

A CPU issued an interrupt to another CPU of a multiprocessor system. We cover such interrupts in the later section "Interprocessor Interrupt Handling."

I/O Interrupt Handling

In general, an I/O interrupt handler must be flexible enough to service several devices at the same time. In the PCI bus architecture, for instance, several devices may share the same IRQ line. This means that the interrupt vector alone does not tell the whole story. In the example shown in Table 4-3, the same vector 43 is assigned to the USB port and to the sound card. However, some hardware devices found in older PC architectures (such as ISA) do not reliably operate if their IRQ line is shared with other devices.

Interrupt handler flexibility is achieved in two distinct ways, as discussed in the fol-lowing list.

IRQ sharing

The interrupt handler executes several *interrupt service routines* (*ISRs*). Each ISR is a function related to a single device sharing the IRQ line. Because it is not pos-sible to know in advance which particular device issued the IRQ, each ISR is exe-cuted to verify whether its device needs attention; if so, the ISR performs all the operations that need to be executed when the device raises an interrupt.

IRQ dynamic allocation

An IRQ line is associated with a device driver at the last possible moment; for instance, the IRQ line of the floppy device is allocated only when a user accesses the floppy disk device. In this way, the same IRQ vector may be used by several hardware devices even if they cannot share the IRQ line; of course, the hardware devices cannot be used at the same time. (See the discussion at the end of this section.)

Not all actions to be performed when an interrupt occurs have the same urgency. In fact, the interrupt handler itself is not a suitable place for all kind of actions. Long noncritical operations should be deferred, because while an interrupt handler is run-ning, the signals on the corresponding IRQ line are temporarily ignored. Most important, the process on behalf of which an interrupt handler is executed must always stay in the TASK_RUNNING state, or a system freeze can occur. Therefore, inter-rupt handlers cannot perform any blocking procedure such as an I/O disk operation. Linux divides the actions to be performed following an interrupt into three classes:

Critical

> Actions such as acknowledging an interrupt to the PIC, reprogramming the PIC or the device controller, or updating data structures accessed by both the device and the processor. These can be executed quickly and are critical, because they must be performed as soon as possible. Critical actions are executed within the interrupt handler immediately, with maskable interrupts disabled.

Noncritical

> Actions such as updating data structures that are accessed only by the processor (for instance, reading the scan code after a keyboard key has been pushed). These actions can also finish quickly, so they are executed by the interrupt handler immediately, with the interrupts enabled.

Noncritical deferrable

> Actions such as copying a buffer's contents into the address space of a process (for instance, sending the keyboard line buffer to the terminal handler process). These may be delayed for a long time interval without affecting the kernel operations; the interested process will just keep waiting for the data. Noncritical deferrable actions are performed by means of separate functions that are discussed in the later section "Softirqs and Tasklets."

Regardless of the kind of circuit that caused the interrupt, all I/O interrupt handlers perform the same four basic actions:

1. Save the IRQ value and the register's contents on the Kernel Mode stack.
2. Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
3. Execute the interrupt service routines (ISRs) associated with all the devices that share the IRQ.
4. Terminate by jumping to the `ret_from_intr( )` address.

Several descriptors are needed to represent both the state of the IRQ lines and the functions to be executed when an interrupt occurs. Figure 4-4 represents in a schematic way the hardware circuits and the software functions used to handle an interrupt. These functions are discussed in the following sections.

Interrupt vectors

As illustrated in Table 4-2, physical IRQs may be assigned any vector in the range 32–238. However, Linux uses vector 128 to implement system calls.

The IBM-compatible PC architecture requires that some devices be statically connected to specific IRQ lines. In particular:

- The interval timer device must be connected to the IRQ0 line (see Chapter 6).
- The slave 8259A PIC must be connected to the IRQ2 line (although more advanced PICs are now being used, Linux still supports 8259A-style PICs).
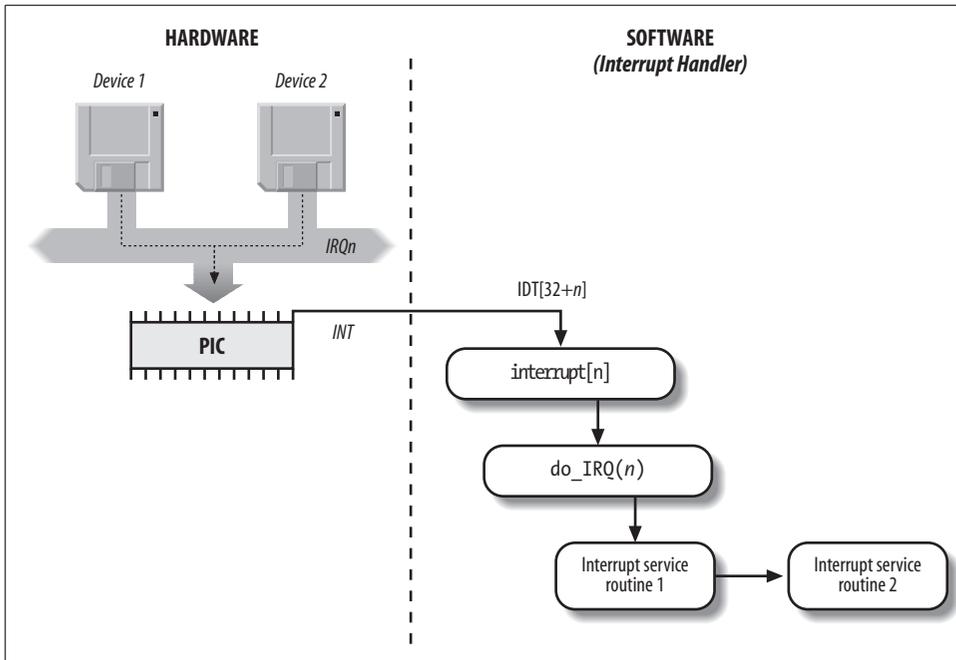
Figure 4-4. I/O interrupt handling

- The external mathematical coprocessor must be connected to the IRQ13 line (although recent 80×86 processors no longer use such a device, Linux continues to support the hardy 80386 model).

- In general, an I/O device can be connected to a limited number of IRQ lines. (As a matter of fact, when playing with an old PC where IRQ sharing is not possible, you might not succeed in installing a new card because of IRQ conflicts with other already present hardware devices.)

Table 4-2. Interrupt vectors in Linux

Vector range	Use
0–19 (0x0–0x13)	Nonmaskable interrupts and exceptions
20–31 (0x14–0x1f)	Intel-reserved
32–127 (0x20–0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls (see Chapter 10)
129–238 (0x81–0xee)	External interrupts (IRQs)
239 (0xef)	Local APIC timer interrupt (see Chapter 6)
240 (0xf0)	Local APIC thermal interrupt (introduced in the Pentium 4 models)
241–250 (0xf1–0xfa)	Reserved by Linux for future use
251–253 (0xfb–0xfd)	Interprocessor interrupts (see the section "Interprocessor Interrupt Handling" later in this chapter)

Table 4-2. Interrupt vectors in Linux (continued)

Vector range	Use
254 (0xfe)	Local APIC error interrupt (generated when the local APIC detects an erroneous condition)
255 (0xff)	Local APIC spurious interrupt (generated if the CPU masks an interrupt while the hardware device raises it)

There are three ways to select a line for an IRQ-configurable device:

- By setting hardware jumpers (only on very old device cards).
- By a utility program shipped with the device and executed when installing it. Such a program may either ask the user to select an available IRQ number or probe the system to determine an available number by itself.
- By a hardware protocol executed at system startup. Peripheral devices declare which interrupt lines they are ready to use; the final values are then negotiated to reduce conflicts as much as possible. Once this is done, each interrupt handler can read the assigned IRQ by using a function that accesses some I/O ports of the device. For instance, drivers for devices that comply with the Peripheral Component Interconnect (PCI) standard use a group of functions such as `pci_read_config_byte( )` to access the device configuration space.

Table 4-3 shows a fairly arbitrary arrangement of devices and IRQs, such as those that might be found on one particular PC.

Table 4-3. An example of IRQ assignment to I/O devices

IRQ	INT	Hardware device
0	32	Timer
1	33	Keyboard
2	34	PIC cascading
3	35	Second serial port
4	36	First serial port
6	38	Floppy disk
8	40	System clock
10	42	Network interface
11	43	USB port, sound card
12	44	PS/2 mouse
13	45	Mathematical coprocessor
14	46	EIDE disk controller's first chain
15	47	EIDE disk controller's second chain

The kernel must discover which I/O device corresponds to the IRQ number before enabling interrupts. Otherwise, for example, how could the kernel handle a signal

from a SCSI disk without knowing which vector corresponds to the device? The correspondence is established while initializing each device driver (see Chapter 13).

IRQ data structures

As always, when discussing complicated operations involving state transitions, it helps to understand first where key data is stored. Thus, this section explains the data structures that support interrupt handling and how they are laid out in various descriptors. Figure 4-5 illustrates schematically the relationships between the main descriptors that represent the state of the IRQ lines. (The figure does not illustrate the data structures needed to handle softirqs and tasklets; they are discussed later in this chapter.)
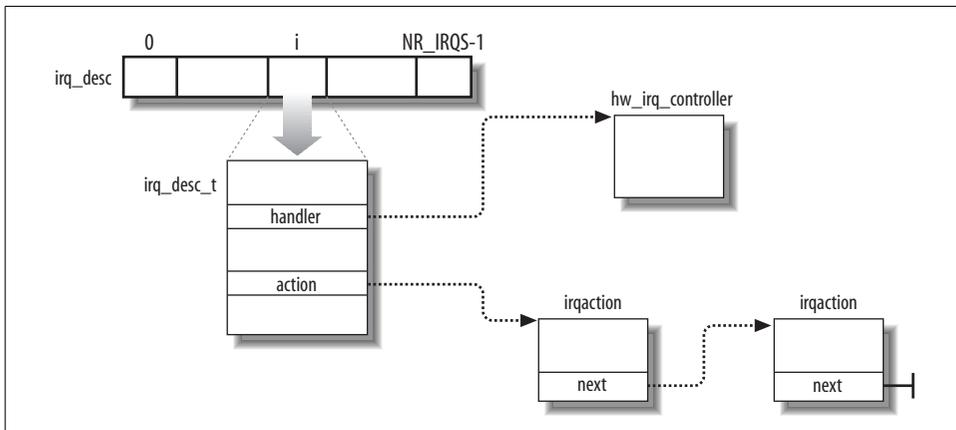
Figure 4-5. IRQ descriptors

Every interrupt vector has its own `irq_desc_t` descriptor, whose fields are listed in Table 4-4. All such descriptors are grouped together in the `irq_desc` array.

Table 4-4. The irq_desc_t descriptor

Field	Description
handler	Points to the PIC object (`hw_irq_controller` descriptor) that services the IRQ line.
handler_data	Pointer to data used by the PIC methods.
action	Identifies the interrupt service routines to be invoked when the IRQ occurs. The field points to the first element of the list of `irqaction` descriptors associated with the IRQ. The `irqaction` descriptor is described later in the chapter.
status	A set of flags describing the IRQ line status (see Table 4-5).
depth	Shows 0 if the IRQ line is enabled and a positive value if it has been disabled at least once.
irq_count	Counter of interrupt occurrences on the IRQ line (for diagnostic use only).
irqs_unhandled	Counter of unhandled interrupt occurrences on the IRQ line (for diagnostic use only).
lock	A spin lock used to serialize the accesses to the IRQ descriptor and to the PIC (see Chapter 5).

An interrupt is *unexpected* if it is not handled by the kernel, that is, either if there is no ISR associated with the IRQ line, or if no ISR associated with the line recognizes the interrupt as raised by its own hardware device. Usually the kernel checks the number of unexpected interrupts received on an IRQ line, so as to disable the line in case a faulty hardware device keeps raising an interrupt over and over. Because the IRQ line can be shared among several devices, the kernel does not disable the line as soon as it detects a single unhandled interrupt. Rather, the kernel stores in the `irq_count` and `irqs_unhandled` fields of the `irq_desc_t` descriptor the total number of interrupts and the number of unexpected interrupts, respectively; when the $100,000^{th}$ interrupt is raised, the kernel disables the line if the number of unhandled interrupts is above 99,900 (that is, if less than 101 interrupts over the last 100,000 received are expected interrupts from hardware devices sharing the line).

The status of an IRQ line is described by the flags listed in Table 4-5.

Table 4-5. Flags describing the IRQ line status

Flag name	Description
IRQ_INPROGRESS	A handler for the IRQ is being executed.
IRQ_DISABLED	The IRQ line has been deliberately disabled by a device driver.
IRQ_PENDING	An IRQ has occurred on the line; its occurrence has been acknowledged to the PIC, but it has not yet been serviced by the kernel.
IRQ_REPLAY	The IRQ line has been disabled but the previous IRQ occurrence has not yet been acknowledged to the PIC.
IRQ_AUTODETECT	The kernel is using the IRQ line while performing a hardware device probe.
IRQ_WAITING	The kernel is using the IRQ line while performing a hardware device probe; moreover, the corresponding interrupt has not been raised.
IRQ_LEVEL	Not used on the 80×86 architecture.
IRQ_MASKED	Not used.
IRQ_PER_CPU	Not used on the 80×86 architecture.

The depth field and the IRQ_DISABLED flag of the `irq_desc_t` descriptor specify whether the IRQ line is enabled or disabled. Every time the `disable_irq()` or `disable_irq_nosync()` function is invoked, the depth field is increased; if depth is equal to 0, the function disables the IRQ line and sets its IRQ_DISABLED flag.[*] Conversely, each invocation of the `enable_irq()` function decreases the field; if depth becomes 0, the function enables the IRQ line and clears its IRQ_DISABLED flag.

During system initialization, the `init_IRQ()` function sets the status field of each IRQ main descriptor to IRQ_DISABLED. Moreover, `init_IRQ()` updates the IDT by replacing the interrupt gates set up by `setup_idt()` (see the section "Preliminary

---

[*] In contrast to `disable_irq_nosync()`, `disable_irq(n)` waits until all interrupt handlers for IRQ n that are running on other CPUs have completed before returning.

Initialization of the IDT," earlier in this chapter) with new ones. This is accomplished through the following statements:

```
for (i = 0; i < NR_IRQS; i++)
    if (i+32 != 128)
        set_intr_gate(i+32,interrupt[i]);
```

This code looks in the `interrupt` array to find the interrupt handler addresses that it uses to set up the interrupt gates. Each entry *n* of the `interrupt` array stores the address of the interrupt handler for IRQ*n* (see the later section "Saving the registers for the interrupt handler"). Notice that the interrupt gate corresponding to vector 128 is left untouched, because it is used for the system call's programmed exception.

In addition to the 8259A chip that was mentioned near the beginning of this chapter, Linux supports several other PIC circuits such as the SMP IO-APIC, Intel PIIX4's internal 8259 PIC, and SGI's Visual Workstation Cobalt (IO-)APIC. To handle all such devices in a uniform way, Linux uses a *PIC object*, consisting of the PIC name and seven PIC standard methods. The advantage of this object-oriented approach is that drivers need not to be aware of the kind of PIC installed in the system. Each driver-visible interrupt source is transparently wired to the appropriate controller. The data structure that defines a PIC object is called `hw_interrupt_type` (also called `hw_irq_controller`).

For the sake of concreteness, let's assume that our computer is a uniprocessor with two 8259A PICs, which provide 16 standard IRQs. In this case, the `handler` field in each of the 16 `irq_desc_t` descriptors points to the `i8259A_irq_type` variable, which describes the 8259A PIC. This variable is initialized as follows:

```
struct hw_interrupt_type i8259A_irq_type = {
    .typename    = "XT-PIC",
    .startup     = startup_8259A_irq,
    .shutdown    = shutdown_8259A_irq,
    .enable      = enable_8259A_irq,
    .disable     = disable_8259A_irq,
    .ack         = mask_and_ack_8259A,
    .end         = end_8259A_irq,
    .set_affinity = NULL
};
```

The first field in this structure, `"XT-PIC"`, is the PIC name. Next come the pointers to six different functions used to program the PIC. The first two functions start up and shut down an IRQ line of the chip, respectively. But in the case of the 8259A chip, these functions coincide with the third and fourth functions, which enable and disable the line. The `mask_and_ack_8259A( )` function acknowledges the IRQ received by sending the proper bytes to the 8259A I/O ports. The `end_8259A_irq( )` function is invoked when the interrupt handler for the IRQ line terminates. The last `set_affinity` method is set to `NULL`: it is used in multiprocessor systems to declare the "affinity" of CPUs for specified IRQs—that is, which CPUs are enabled to handle specific IRQs.

As described earlier, multiple devices can share a single IRQ. Therefore, the kernel maintains `irqaction` descriptors (see Figure 4-5 earlier in this chapter), each of which refers to a specific hardware device and a specific interrupt. The fields included in such descriptor are shown in Table 4-6, and the flags are shown in Table 4-7.

Table 4-6. Fields of the irqaction descriptor

Field name	Description
handler	Points to the interrupt service routine for an I/O device. This is the key field that allows many devices to share the same IRQ.
flags	This field includes a few fields that describe the relationships between the IRQ line and the I/O device (see Table 4-7).
mask	Not used.
name	The name of the I/O device (shown when listing the serviced IRQs by reading the */proc/interrupts* file).
dev_id	A private field for the I/O device. Typically, it identifies the I/O device itself (for instance, it could be equal to its major and minor numbers; see the section "Device Files" in Chapter 13), or it points to the device driver's data.
next	Points to the next element of a list of `irqaction` descriptors. The elements in the list refer to hardware devices that share the same IRQ.
irq	IRQ line.
dir	Points to the descriptor of the */proc/irq/n* directory associated with the IRQ*n*.

Table 4-7. Flags of the irqaction descriptor

Flag name	Description
SA_INTERRUPT	The handler must execute with interrupts disabled.
SA_SHIRQ	The device permits its IRQ line to be shared with other devices.
SA_SAMPLE_RANDOM	The device may be considered a source of events that occurs randomly; it can thus be used by the kernel random number generator. (Users can access this feature by taking random numbers from the */dev/random* and */dev/urandom* device files.)

Finally, the `irq_stat` array includes NR_CPUS entries, one for every possible CPU in the system. Each entry of type `irq_cpustat_t` includes a few counters and flags used by the kernel to keep track of what each CPU is currently doing (see Table 4-8).

Table 4-8. Fields of the irq_cpustat_t structure

Field name	Description
__softirq_pending	Set of flags denoting the pending softirqs (see the section "Softirqs" later in this chapter)
idle_timestamp	Time when the CPU became idle (significant only if the CPU is currently idle)
__nmi_count	Number of occurrences of NMI interrupts
apic_timer_irqs	Number of occurrences of local APIC timer interrupts (see Chapter 6)

IRQ distribution in multiprocessor systems

Linux sticks to the Symmetric Multiprocessing model (SMP); this means, essentially, that the kernel should not have any bias toward one CPU with respect to the others.

As a consequence, the kernel tries to distribute the IRQ signals coming from the hardware devices in a round-robin fashion among all the CPUs. Therefore, all the CPUs should spend approximately the same fraction of their execution time servicing I/O interrupts.

In the earlier section "The Advanced Programmable Interrupt Controller (APIC)," we said that the multi-APIC system has sophisticated mechanisms to dynamically distribute the IRQ signals among the CPUs.

During system bootstrap, the booting CPU executes the `setup_IO_APIC_irqs()` function to initialize the I/O APIC chip. The 24 entries of the Interrupt Redirection Table of the chip are filled, so that all IRQ signals from the I/O hardware devices can be routed to each CPU in the system according to the "lowest priority" scheme (see the earlier section "IRQs and Interrupts"). During system bootstrap, moreover, all CPUs execute the `setup_local_APIC()` function, which takes care of initializing the local APICs. In particular, the task priority register (TPR) of each chip is initialized to a fixed value, meaning that the CPU is willing to handle every kind of IRQ signal, regardless of its priority. The Linux kernel never modifies this value after its initialization.

All task priority registers contain the same value, thus all CPUs always have the same priority. To break a tie, the multi-APIC system uses the values in the arbitration priority registers of local APICs, as explained earlier. Because such values are automatically changed after every interrupt, the IRQ signals are, in most cases, fairly distributed among all CPUs.[*]

In short, when a hardware device raises an IRQ signal, the multi-APIC system selects one of the CPUs and delivers the signal to the corresponding local APIC, which in turn interrupts its CPU. No other CPUs are notified of the event.

All this is magically done by the hardware, so it should be of no concern for the kernel after multi-APIC system initialization. Unfortunately, in some cases the hardware fails to distribute the interrupts among the microprocessors in a fair way (for instance, some Pentium 4–based SMP motherboards have this problem). Therefore, Linux 2.6 makes use of a special kernel thread called *kirqd* to correct, if necessary, the automatic assignment of IRQs to CPUs.

The kernel thread exploits a nice feature of multi-APIC systems, called the *IRQ affinity* of a CPU: by modifying the Interrupt Redirection Table entries of the I/O APIC, it is possible to route an interrupt signal to a specific CPU. This can be done by invoking the `set_ioapic_affinity_irq()` function, which acts on two parameters: the IRQ vector to be rerouted and a 32-bit mask denoting the CPUs that can receive the IRQ. The IRQ affinity of a given interrupt also can be changed by the system administra-

---

[*] There is an exception, though. Linux usually sets up the local APICs in such a way to honor the *focus processor*, when it exists. A focus process will catch all IRQs of the same type as long as it has received an IRQ of that type, and it has not finished executing the interrupt handler. However, Intel has dropped support for focus processors in the Pentium 4 model.

tor by writing a new CPU bitmap mask into the */proc/irq/n/smp_affinity* file (*n* being the interrupt vector).

The *kirqd* kernel thread periodically executes the `do_irq_balance()` function, which keeps track of the number of interrupt occurrences received by every CPU in the most recent time interval. If the function discovers that the IRQ load imbalance between the heaviest loaded CPU and the least loaded CPU is significantly high, then it either selects an IRQ to be "moved" from a CPU to another, or rotates all IRQs among all existing CPUs.

Multiple Kernel Mode stacks

As mentioned in the section "Identifying a Process" in Chapter 3, the `thread_info` descriptor of each process is coupled with a Kernel Mode stack in a `thread_union` data structure composed by one or two page frames, according to an option selected when the kernel has been compiled. If the size of the `thread_union` structure is 8 KB, the Kernel Mode stack of the current process is used for every type of kernel control path: exceptions, interrupts, and deferrable functions (see the later section "Softirqs and Tasklets"). Conversely, if the size of the `thread_union` structure is 4 KB, the kernel makes use of three types of Kernel Mode stacks:

- The *exception stack* is used when handling exceptions (including system calls). This is the stack contained in the per-process `thread_union` data structure, thus the kernel makes use of a different exception stack for each process in the system.

- The *hard IRQ stack* is used when handling interrupts. There is one hard IRQ stack for each CPU in the system, and each stack is contained in a single page frame.

- The *soft IRQ stack* is used when handling deferrable functions (softirqs or tasklets; see the later section "Softirqs and Tasklets"). There is one soft IRQ stack for each CPU in the system, and each stack is contained in a single page frame.

All hard IRQ stacks are contained in the `hardirq_stack` array, while all soft IRQ stacks are contained in the `softirq_stack` array. Each array element is a union of type `irq_ctx` that span a single page. At the bottom of this page is stored a `thread_info` structure, while the spare memory locations are used for the stack; remember that each stack grows towards lower addresses. Thus, hard IRQ stacks and soft IRQ stacks are very similar to the exception stacks described in the section "Identifying a Process" in Chapter 3; the only difference is that the `thread_info` structure coupled with each stack is associated with a CPU rather than a process.

The `hardirq_ctx` and `softirq_ctx` arrays allow the kernel to quickly determine the hard IRQ stack and soft IRQ stack of a given CPU, respectively: they contain pointers to the corresponding `irq_ctx` elements.

Saving the registers for the interrupt handler

When a CPU receives an interrupt, it starts executing the code at the address found in the corresponding gate of the IDT (see the earlier section "Hardware Handling of Interrupts and Exceptions").

As with other context switches, the need to save registers leaves the kernel developer with a somewhat messy coding job, because the registers have to be saved and restored using assembly language code. However, within those operations, the processor is expected to call and return from a C function. In this section, we describe the assembly language task of handling registers; in the next, we show some of the acrobatics required in the C function that is subsequently invoked.

Saving registers is the first task of the interrupt handler. As already mentioned, the address of the interrupt handler for IRQ n is initially stored in the interrupt[n] entry and then copied into the interrupt gate included in the proper IDT entry.

The interrupt array is built through a few assembly language instructions in the *arch/i386/kernel/entry.S* file. The array includes NR_IRQS elements, where the NR_IRQS macro yields either the number 224 if the kernel supports a recent I/O APIC chip,[*] or the number 16 if the kernel uses the older 8259A PIC chips. The element at index n in the array stores the address of the following two assembly language instructions:

```
pushl $n-256
jmp common_interrupt
```

The result is to save on the stack the IRQ number associated with the interrupt minus 256. The kernel represents all IRQs through negative numbers, because it reserves positive interrupt numbers to identify system calls (see Chapter 10). The same code for all interrupt handlers can then be executed while referring to this number. The common code starts at label common_interrupt and consists of the following assembly language macros and instructions:

```
common_interrupt:
    SAVE_ALL
    movl %esp,%eax
    call do_IRQ
    jmp ret_from_intr
```

The SAVE_ALL macro expands to the following fragment:

```
cld
push %es
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
```

---

[*] 256 vectors is an architectural limit for the 80×86 architecture. 32 of them are used or reserved for the CPU, so the usable vector space consists of 224 vectors.

```
pushl %edx
pushl %ecx
pushl %ebx
movl $__USER_DS,%edx
movl %edx,%ds
movl %edx,%es
```

SAVE_ALL saves all the CPU registers that may be used by the interrupt handler on the stack, except for eflags, cs, eip, ss, and esp, which are already saved automatically by the control unit (see the earlier section "Hardware Handling of Interrupts and Exceptions"). The macro then loads the selector of the user data segment into ds and es.

After saving the registers, the address of the current top stack location is saved in the eax register; then, the interrupt handler invokes the do_IRQ() function. When the ret instruction of do_IRQ() is executed (when that function terminates) control is transferred to ret_from_intr() (see the later section "Returning from Interrupts and Exceptions").

The do_IRQ( ) function

The do_IRQ() function is invoked to execute all interrupt service routines associated with an interrupt. It is declared as follows:

```
__attribute__((regparm(3))) unsigned int do_IRQ(struct pt_regs *regs)
```

The regparm keyword instructs the function to go to the eax register to find the value of the regs argument; as seen above, eax points to the stack location containing the last register value pushed on by SAVE_ALL.

The do_IRQ() function executes the following actions:

1. Executes the irq_enter() macro, which increases a counter representing the number of nested interrupt handlers. The counter is stored in the preempt_count field of the thread_info structure of the current process (see Table 4-10 later in this chapter).

2. If the size of the thread_union structure is 4 KB, it switches to the hard IRQ stack.In particular, the function performs the following substeps:

   a. Executes the current_thread_info() function to get the address of the thread_info descriptor associated with the Kernel Mode stack addressed by the esp register (see the section "Identifying a Process" in Chapter 3).

   b. Compares the address of the thread_info descriptor obtained in the previous step with the address stored in hardirq_ctx[smp_processor_id()], that is, the address of the thread_info descriptor associated with the local CPU. If the two addresses are equal, the kernel is already using the hard IRQ stack, thus jumps to step 3. This happens when an IRQ is raised while the kernel is still handling another interrupt.

   c. Here the Kernel Mode stack has to be switched. Stores the pointer to the current process descriptor in the task field of the thread_info descriptor in

irq_ctx union of the local CPU. This is done so that the current macro works as expected while the kernel is using the hard IRQ stack (see the section "Identifying a Process" in Chapter 3).

    d. Stores the current value of the esp stack pointer register in the previous_esp field of the thread_info descriptor in the irq_ctx union of the local CPU (this field is used only when preparing the function call trace for a kernel oops).

    e. Loads in the esp stack register the top location of the hard IRQ stack of the local CPU (the value in hardirq_ctx[smp_processor_id()] plus 4096); the previous value of the esp register is saved in the ebx register.

3. Invokes the __do_IRQ() function passing to it the pointer regs and the IRQ number obtained from the regs->orig_eax field (see the following section).

4. If the hard IRQ stack has been effectively switched in step 2e above, the function copies the original stack pointer from the ebx register into the esp register, thus switching back to the exception stack or soft IRQ stack that were in use before.

5. Executes the irq_exit() macro, which decreases the interrupt counter and checks whether deferrable kernel functions are waiting to be executed (see the section "Softirqs and Tasklets" later in this chapter).

6. Terminates: the control is transferred to the ret_from_intr() function (see the later section "Returning from Interrupts and Exceptions").

The __do_IRQ() function

The __do_IRQ() function receives as its parameters an IRQ number (through the eax register) and a pointer to the pt_regs structure where the User Mode register values have been saved (through the edx register).

The function is equivalent to the following code fragment:

```
spin_lock(&(irq_desc[irq].lock));
irq_desc[irq].handler->ack(irq);
irq_desc[irq].status &= ~(IRQ_REPLAY | IRQ_WAITING);
irq_desc[irq].status |= IRQ_PENDING;
if (!(irq_desc[irq].status & (IRQ_DISABLED | IRQ_INPROGRESS))
        && irq_desc[irq].action) {
    irq_desc[irq].status |= IRQ_INPROGRESS;
    do {
        irq_desc[irq].status &= ~IRQ_PENDING;
        spin_unlock(&(irq_desc[irq].lock));
        handle_IRQ_event(irq, regs, irq_desc[irq].action);
        spin_lock(&(irq_desc[irq].lock));
    } while (irq_desc[irq].status & IRQ_PENDING);
    irq_desc[irq].status &= ~IRQ_INPROGRESS;
}
irq_desc[irq].handler->end(irq);
spin_unlock(&(irq_desc[irq].lock));
```

Before accessing the main IRQ descriptor, the kernel acquires the corresponding spin lock. We'll see in Chapter 5 that the spin lock protects against concurrent accesses

by different CPUs. This spin lock is necessary in a multiprocessor system, because other interrupts of the same kind may be raised, and other CPUs might take care of the new interrupt occurrences. Without the spin lock, the main IRQ descriptor would be accessed concurrently by several CPUs. As we'll see, this situation must be absolutely avoided.

After acquiring the spin lock, the function invokes the `ack` method of the main IRQ descriptor. When using the old 8259A PIC, the corresponding `mask_and_ack_8259A()` function acknowledges the interrupt on the PIC and also disables the IRQ line. Masking the IRQ line ensures that the CPU does not accept further occurrences of this type of interrupt until the handler terminates. Remember that the `__do_IRQ()` function runs with local interrupts disabled; in fact, the CPU control unit automatically clears the `IF` flag of the `eflags` register because the interrupt handler is invoked through an IDT's interrupt gate. However, we'll see shortly that the kernel might re-enable local interrupts before executing the interrupt service routines of this interrupt.

When using the I/O APIC, however, things are much more complicated. Depending on the type of interrupt, acknowledging the interrupt could either be done by the `ack` method or delayed until the interrupt handler terminates (that is, acknowledgement could be done by the `end` method). In either case, we can take for granted that the local APIC doesn't accept further interrupts of this type until the handler terminates, although further occurrences of this type of interrupt may be accepted by other CPUs.

The `__do_IRQ()` function then initializes a few flags of the main IRQ descriptor. It sets the `IRQ_PENDING` flag because the interrupt has been acknowledged (well, sort of), but not yet really serviced; it also clears the `IRQ_WAITING` and `IRQ_REPLAY` flags (but we don't have to care about them now).

Now `__do_IRQ()` checks whether it must really handle the interrupt. There are three cases in which nothing has to be done. These are discussed in the following list.

`IRQ_DISABLED` *is set*

> A CPU might execute the `__do_IRQ()` function even if the corresponding IRQ line is disabled; you'll find an explanation for this nonintuitive case in the later section "Reviving a lost interrupt." Moreover, buggy motherboards may generate spurious interrupts even when the IRQ line is disabled in the PIC.

`IRQ_INPROGRESS` *is set*

> In a multiprocessor system, another CPU might be handling a previous occurrence of the same interrupt. Why not defer the handling of *this* occurrence to *that* CPU? This is exactly what is done by Linux. This leads to a simpler kernel architecture because device drivers' interrupt service routines need not to be reentrant (their execution is serialized). Moreover, the freed CPU can quickly return to what it was doing, without dirtying its hardware cache; this is beneficial to system performance. The `IRQ_INPROGRESS` flag is set whenever a CPU is committed to execute the interrupt service routines of the interrupt; therefore, the `__do_IRQ()` function checks it before starting the real work.

`irq_desc[irq].action` *is* `NULL`

> This case occurs when there is no interrupt service routine associated with the interrupt. Normally, this happens only when the kernel is probing a hardware device.

Let's suppose that none of the three cases holds, so the interrupt has to be serviced. The `__do_IRQ()` function sets the `IRQ_INPROGRESS` flag and starts a loop. In each iteration, the function clears the `IRQ_PENDING` flag, releases the interrupt spin lock, and executes the interrupt service routines by invoking `handle_IRQ_event()` (described later in the chapter). When the latter function terminates, `__do_IRQ()` acquires the spin lock again and checks the value of the `IRQ_PENDING` flag. If it is clear, no further occurrence of the interrupt has been delivered to another CPU, so the loop ends. Conversely, if `IRQ_PENDING` is set, another CPU has executed the `do_IRQ()` function for this type of interrupt while this CPU was executing `handle_IRQ_event()`. Therefore, `do_IRQ()` performs another iteration of the loop, servicing the new occurrence of the interrupt.[*]

Our `__do_IRQ()` function is now going to terminate, either because it has already executed the interrupt service routines or because it had nothing to do. The function invokes the `end` method of the main IRQ descriptor. When using the old 8259A PIC, the corresponding `end_8259A_irq()` function reenables the IRQ line (unless the interrupt occurrence was spurious). When using the I/O APIC, the `end` method acknowledges the interrupt (if not already done by the `ack` method).

Finally, `__do_IRQ()` releases the spin lock: the hard work is finished!

Reviving a lost interrupt

The `__do_IRQ()` function is small and simple, yet it works properly in most cases. Indeed, the `IRQ_PENDING`, `IRQ_INPROGRESS`, and `IRQ_DISABLED` flags ensure that interrupts are correctly handled even when the hardware is misbehaving. However, things may not work so smoothly in a multiprocessor system.

Suppose that a CPU has an IRQ line enabled. A hardware device raises the IRQ line, and the multi-APIC system selects our CPU for handling the interrupt. Before the CPU acknowledges the interrupt, the IRQ line is masked out by another CPU; as a consequence, the `IRQ_DISABLED` flag is set. Right afterwards, our CPU starts handling the pending interrupt; therefore, the `do_IRQ()` function acknowledges the interrupt and then returns without executing the interrupt service routines because it finds the `IRQ_DISABLED` flag set. Therefore, even though the interrupt occurred before the IRQ line was disabled, it gets lost.

To cope with this scenario, the `enable_irq()` function, which is used by the kernel to enable an IRQ line, checks first whether an interrupt has been lost. If so, the function forces the hardware to generate a new occurrence of the lost interrupt:

---

[*] Because `IRQ_PENDING` is a flag and not a counter, only the second occurrence of the interrupt can be recognized. Further occurrences in each iteration of the `do_IRQ()`'s loop are simply lost.

```
    spin_lock_irqsave(&(irq_desc[irq].lock), flags);
    if (--irq_desc[irq].depth == 0) {
        irq_desc[irq].status &= ~IRQ_DISABLED;
        if (irq_desc[irq].status & (IRQ_PENDING | IRQ_REPLAY))
                == IRQ_PENDING) {
            irq_desc[irq].status |= IRQ_REPLAY;
            hw_resend_irq(irq_desc[irq].handler,irq);
        }
        irq_desc[irq].handler->enable(irq);
    }
    spin_lock_irqrestore(&(irq_desc[irq].lock), flags);
```

The function detects that an interrupt was lost by checking the value of the IRQ_PENDING flag. The flag is always cleared when leaving the interrupt handler; therefore, if the IRQ line is disabled and the flag is set, then an interrupt occurrence has been acknowledged but not yet serviced. In this case the hw_resend_irq() function raises a new interrupt. This is obtained by forcing the local APIC to generate a self-interrupt (see the later section "Interprocessor Interrupt Handling"). The role of the IRQ_REPLAY flag is to ensure that exactly one self-interrupt is generated. Remember that the __do_IRQ() function clears that flag when it starts handling the interrupt.

Interrupt service routines

As mentioned previously, an interrupt service routine handles an interrupt by executing an operation specific to one type of device. When an interrupt handler must execute the ISRs, it invokes the handle_IRQ_event() function. This function essentially performs the following steps:

1. Enables the local interrupts with the sti assembly language instruction if the SA_INTERRUPT flag is clear.

2. Executes each interrupt service routine of the interrupt through the following code:

   ```
   retval = 0;
   do {
       retval |= action->handler(irq, action->dev_id, regs);
       action = action->next;
   } while (action);
   ```

   At the start of the loop, action points to the start of a list of irqaction data structures that indicate the actions to be taken upon receiving the interrupt (see Figure 4-5 earlier in this chapter).

3. Disables local interrupts with the cli assembly language instruction.

4. Terminates by returning the value of the retval local variable, that is, 0 if no interrupt service routine has recognized interrupt, 1 otherwise (see next).

All interrupt service routines act on the same parameters (once again they are passed through the eax, edx, and ecx registers, respectively):

irq

> The IRQ number

dev_id

> The device identifier

regs

> A pointer to a `pt_regs` structure on the Kernel Mode (exception) stack containing the registers saved right after the interrupt occurred. The `pt_regs` structure consists of 15 fields:
>
>   • The first nine fields are the register values pushed by `SAVE_ALL`
>
>   • The tenth field, referenced through a field called `orig_eax`, encodes the IRQ number
>
>   • The remaining fields correspond to the register values pushed on automatically by the control unit

The first parameter allows a single ISR to handle several IRQ lines, the second one allows a single ISR to take care of several devices of the same type, and the last one allows the ISR to access the execution context of the interrupted kernel control path. In practice, most ISRs do not use these parameters.

Every interrupt service routine returns the value 1 if the interrupt has been effectively handled, that is, if the signal was raised by the hardware device handled by the interrupt service routine (and not by another device sharing the same IRQ); it returns the value 0 otherwise. This return code allows the kernel to update the counter of unexpected interrupts mentioned in the section "IRQ data structures" earlier in this chapter.

The `SA_INTERRUPT` flag of the main IRQ descriptor determines whether interrupts must be enabled or disabled when the `do_IRQ( )` function invokes an ISR. An ISR that has been invoked with the interrupts in one state is allowed to put them in the opposite state. In a uniprocessor system, this can be achieved by means of the `cli` (disable interrupts) and `sti` (enable interrupts) assembly language instructions.

The structure of an ISR depends on the characteristics of the device handled. We'll give a couple of examples of ISRs in Chapter 6 and Chapter 13.

Dynamic allocation of IRQ lines

As noted in section "Interrupt vectors," a few vectors are reserved for specific devices, while the remaining ones are dynamically handled. There is, therefore, a way in which the same IRQ line can be used by several hardware devices even if they do not allow IRQ sharing. The trick is to serialize the activation of the hardware devices so that just one owns the IRQ line at a time.

Before activating a device that is going to use an IRQ line, the corresponding driver invokes request_irq( ). This function creates a new irqaction descriptor and initializes it with the parameter values; it then invokes the setup_irq( ) function to insert the descriptor in the proper IRQ list. The device driver aborts the operation if setup_irq( ) returns an error code, which usually means that the IRQ line is already in use by another device that does not allow interrupt sharing. When the device operation is concluded, the driver invokes the free_irq( ) function to remove the descriptor from the IRQ list and release the memory area.

Let's see how this scheme works on a simple example. Assume a program wants to address the */dev/fd0* device file, which corresponds to the first floppy disk on the system.* The program can do this either by directly accessing */dev/fd0* or by mounting a filesystem on it. Floppy disk controllers are usually assigned IRQ6; given this, a floppy driver may issue the following request:

```
request_irq(6, floppy_interrupt,
            SA_INTERRUPT|SA_SAMPLE_RANDOM, "floppy", NULL);
```

As can be observed, the floppy_interrupt( ) interrupt service routine must execute with the interrupts disabled (SA_INTERRUPT flag set) and no sharing of the IRQ (SA_SHIRQ flag missing). The SA_SAMPLE_RANDOM flag set means that accesses to the floppy disk are a good source of random events to be used for the kernel random number generator. When the operation on the floppy disk is concluded (either the I/O operation on */dev/fd0* terminates or the filesystem is unmounted), the driver releases IRQ6:

```
free_irq(6, NULL);
```

To insert an irqaction descriptor in the proper list, the kernel invokes the setup_irq( ) function, passing to it the parameters irq_nr, the IRQ number, and new (the address of a previously allocated irqaction descriptor). This function:

1. Checks whether another device is already using the irq_nr IRQ and, if so, whether the SA_SHIRQ flags in the irqaction descriptors of both devices specify that the IRQ line can be shared. Returns an error code if the IRQ line cannot be used.

2. Adds *new (the new irqaction descriptor pointed to by new) at the end of the list to which irq_desc[irq_nr]->action points.

3. If no other device is sharing the same IRQ, the function clears the IRQ_DISABLED, IRQ_AUTODETECT, IRQ_WAITING, and IRQ_INPROGRESS flags in the flags field of *new and invokes the startup method of the irq_desc[irq_nr]->handler PIC object to make sure that IRQ signals are enabled.

Here is an example of how setup_irq( ) is used, drawn from system initialization. The kernel initializes the irq0 descriptor of the interval timer device by executing the following instructions in the time_init( ) function (see Chapter 6):

---

* Floppy disks are "old" devices that do not usually allow IRQ sharing.

```
struct irqaction irq0 =
    {timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
setup_irq(0, &irq0);
```

First, the irq0 variable of type irqaction is initialized: the handler field is set to the address of the timer_interrupt( ) function, the flags field is set to SA_INTERRUPT, the name field is set to *"timer"*, and the fifth field is set to NULL to show that no dev_id value is used. Next, the kernel invokes setup_irq( ) to insert irq0 in the list of irqaction descriptors associated with IRQ0.

Interprocessor Interrupt Handling

Interprocessor interrupts allow a CPU to send interrupt signals to any other CPU in the system. As explained in the section "The Advanced Programmable Interrupt Controller (APIC)" earlier in this chapter, an interprocessor interrupt (IPI) is delivered not through an IRQ line, but directly as a message on the bus that connects the local APIC of all CPUs (either a dedicated bus in older motherboards, or the system bus in the Pentium 4–based motherboards).

On multiprocessor systems, Linux makes use of three kinds of interprocessor interrupts (see also Table 4-2):

CALL_FUNCTION_VECTOR *(vector* 0xfb*)*
> Sent to all CPUs but the sender, forcing those CPUs to run a function passed by the sender. The corresponding interrupt handler is named call_function_ interrupt( ). The function (whose address is passed in the call_data global variable) may, for instance, force all other CPUs to stop, or may force them to set the contents of the Memory Type Range Registers (MTRRs).[*] Usually this interrupt is sent to all CPUs except the CPU executing the calling function by means of the smp_call_function( ) facility function.

RESCHEDULE_VECTOR *(vector* 0xfc*)*
> When a CPU receives this type of interrupt, the corresponding handler—named reschedule_interrupt( )—limits itself to acknowledging the interrupt. Rescheduling is done automatically when returning from the interrupt (see the section "Returning from Interrupts and Exceptions" later in this chapter).

INVALIDATE_TLB_VECTOR *(vector* 0xfd*)*
> Sent to all CPUs but the sender, forcing them to invalidate their Translation Lookaside Buffers. The corresponding handler, named invalidate_interrupt( ), flushes some TLB entries of the processor as described in the section "Handling the Hardware Cache and the TLB" in Chapter 2.

---

[*] Starting with the Pentium Pro model, Intel microprocessors include these additional registers to easily customize cache operations. For instance, Linux may use these registers to disable the hardware cache for the addresses mapping the frame buffer of a PCI/AGP graphic card while maintaining the "write combining" mode of operation: the paging unit combines write transfers into larger chunks before copying them into the frame buffer.

The assembly language code of the interprocessor interrupt handlers is generated by the `BUILD_INTERRUPT` macro: it saves the registers, pushes the vector number minus 256 on the stack, and then invokes a high-level C function having the same name as the low-level handler preceded by `smp_`. For instance, the high-level handler of the `CALL_FUNCTION_VECTOR` interprocessor interrupt that is invoked by the low-level `call_function_interrupt()` handler is named `smp_call_function_interrupt()`. Each high-level handler acknowledges the interprocessor interrupt on the local APIC and then performs the specific action triggered by the interrupt.

Thanks to the following group of functions, issuing interprocessor interrupts (IPIs) becomes an easy task:

`send_IPI_all( )`
> Sends an IPI to all CPUs (including the sender)

`send_IPI_allbutself( )`
> Sends an IPI to all CPUs except the sender

`send_IPI_self( )`
> Sends an IPI to the sender CPU

`send_IPI_mask( )`
> Sends an IPI to a group of CPUs specified by a bit mask

Softirqs and Tasklets

We mentioned earlier in the section "Interrupt Handling" that several tasks among those executed by the kernel are not critical: they can be deferred for a long period of time, if necessary. Remember that the interrupt service routines of an interrupt handler are serialized, and often there should be no occurrence of an interrupt until the corresponding interrupt handler has terminated. Conversely, the deferrable tasks can execute with all interrupts enabled. Taking them out of the interrupt handler helps keep kernel response time small. This is a very important property for many time-critical applications that expect their interrupt requests to be serviced in a few milliseconds.

Linux 2.6 answers such a challenge by using two kinds of non-urgent interruptible kernel functions: the so-called *deferrable functions*[*] (*softirqs* and *tasklets*), and those executed by means of some work queues (we will describe them in the section "Work Queues" later in this chapter).

Softirqs and tasklets are strictly correlated, because tasklets are implemented on top of softirqs. As a matter of fact, the term "softirq," which appears in the kernel source code, often denotes both kinds of deferrable functions. Another widely used term is

---

[*] These are also called *software interrupt*s, but we denote them as "deferrable functions" to avoid confusion with programmed exceptions, which are referred to as "software interrupts" in Intel manuals.

interrupt context: it specifies that the kernel is currently executing either an interrupt handler or a deferrable function.

Softirqs are statically allocated (i.e., defined at compile time), while tasklets can also be allocated and initialized at runtime (for instance, when loading a kernel module). Softirqs can run concurrently on several CPUs, even if they are of the same type. Thus, softirqs are reentrant functions and must explicitly protect their data structures with spin locks. Tasklets do not have to worry about this, because their execution is controlled more strictly by the kernel. Tasklets of the same type are always serialized: in other words, the same type of tasklet cannot be executed by two CPUs at the same time. However, tasklets of different types can be executed concurrently on several CPUs. Serializing the tasklet simplifies the life of device driver developers, because the tasklet function needs not be reentrant.

Generally speaking, four kinds of operations can be performed on deferrable functions:

Initialization

> Defines a new deferrable function; this operation is usually done when the kernel initializes itself or a module is loaded.

Activation

> Marks a deferrable function as "pending"—to be run the next time the kernel schedules a round of executions of deferrable functions. Activation can be done at any time (even while handling interrupts).

Masking

> Selectively disables a deferrable function so that it will not be executed by the kernel even if activated. We'll see in the section "Disabling and Enabling Deferrable Functions" in Chapter 5 that disabling deferrable functions is sometimes essential.

Execution

> Executes a pending deferrable function together with all other pending deferrable functions of the same type; execution is performed at well-specified times, explained later in the section "Softirqs."

Activation and execution are bound together: a deferrable function that has been activated by a given CPU must be executed on the same CPU. There is no self-evident reason suggesting that this rule is beneficial for system performance. Binding the deferrable function to the activating CPU could in theory make better use of the CPU hardware cache. After all, it is conceivable that the activating kernel thread accesses some data structures that will also be used by the deferrable function. However, the relevant lines could easily be no longer in the cache when the deferrable function is run because its execution can be delayed a long time. Moreover, binding a function to a CPU is always a potentially "dangerous" operation, because one CPU might end up very busy while the others are mostly idle.

Softirqs

Linux 2.6 uses a limited number of softirqs. For most purposes, tasklets are good enough and are much easier to write because they do not need to be reentrant.

As a matter of fact, only the six kinds of softirqs listed in Table 4-9 are currently defined.

Table 4-9. Softirqs used in Linux 2.6

Softirq	Index (priority)	Description
HI_SOFTIRQ	0	Handles high priority tasklets
TIMER_SOFTIRQ	1	Tasklets related to timer interrupts
NET_TX_SOFTIRQ	2	Transmits packets to network cards
NET_RX_SOFTIRQ	3	Receives packets from network cards
SCSI_SOFTIRQ	4	Post-interrupt processing of SCSI commands
TASKLET_SOFTIRQ	5	Handles regular tasklets

The index of a sofirq determines its priority: a lower index means higher priority because softirq functions will be executed starting from index 0.

Data structures used for softirqs

The main data structure used to represent softirqs is the softirq_vec array, which includes 32 elements of type softirq_action. The priority of a softirq is the index of the corresponding softirq_action element inside the array. As shown in Table 4-9, only the first six entries of the array are effectively used. The softirq_action data structure consists of two fields: an action pointer to the softirq function and a data pointer to a generic data structure that may be needed by the softirq function.

Another critical field used to keep track both of kernel preemption and of nesting of kernel control paths is the 32-bit preempt_count field stored in the thread_info field of each process descriptor (see the section "Identifying a Process" in Chapter 3). This field encodes three distinct counters plus a flag, as shown in Table 4-10.

Table 4-10. Subfields of the preempt_count field (continues)

Bits	Description
0–7	Preemption counter (max value = 255)
8–15	Softirq counter (max value = 255).
16–27	Hardirq counter (max value = 4096)
28	PREEMPT_ACTIVE flag

The first counter keeps track of how many times kernel preemption has been explicitly disabled on the local CPU; the value zero means that kernel preemption has not

been explicitly disabled at all. The second counter specifies how many levels deep the disabling of deferrable functions is (level 0 means that deferrable functions are enabled). The third counter specifies the number of nested interrupt handlers on the local CPU (the value is increased by `irq_enter()` and decreased by `irq_exit()`; see the section "I/O Interrupt Handling" earlier in this chapter).

There is a good reason for the name of the `preempt_count` field: kernel preemptability has to be disabled either when it has been explicitly disabled by the kernel code (preemption counter not zero) or when the kernel is running in interrupt context. Thus, to determine whether the current process can be preempted, the kernel quickly checks for a zero value in the `preempt_count` field. Kernel preemption will be discussed in depth in the section "Kernel Preemption" in Chapter 5.

The `in_interrupt()` macro checks the hardirq and softirq counters in the `current_thread_info()->preempt_count` field. If either one of these two counters is positive, the macro yields a nonzero value, otherwise it yields the value zero. If the kernel does not make use of multiple Kernel Mode stacks, the macro always looks at the `preempt_count` field of the `thread_info` descriptor of the current process. If, however, the kernel makes use of multiple Kernel Mode stacks, the macro might look at the `preempt_count` field in the `thread_info` descriptor contained in a `irq_ctx` union associated with the local CPU. In this case, the macro returns a nonzero value because the field is always set to a positive value.

The last crucial data structure for implementing the softirqs is a per-CPU 32-bit mask describing the pending softirqs; it is stored in the `__softirq_pending` field of the `irq_cpustat_t` data structure (recall that there is one such structure per each CPU in the system; see Table 4-8). To get and set the value of the bit mask, the kernel makes use of the `local_softirq_pending()` macro that selects the softirq bit mask of the local CPU.

Handling softirqs

The `open_softirq()` function takes care of softirq initialization. It uses three parameters: the softirq index, a pointer to the softirq function to be executed, and a second pointer to a data structure that may be required by the softirq function. `open_softirq()` limits itself to initializing the proper entry of the `softirq_vec` array.

Softirqs are activated by means of the `raise_softirq()` function. This function, which receives as its parameter the softirq index `nr`, performs the following actions:

1. Executes the `local_irq_save` macro to save the state of the IF flag of the `eflags` register and to disable interrupts on the local CPU.

2. Marks the softirq as pending by setting the bit corresponding to the index `nr` in the softirq bit mask of the local CPU.

3. If `in_interrupt()` yields the value 1, it jumps to step 5. This situation indicates either that `raise_softirq()` has been invoked in interrupt context, or that the softirqs are currently disabled.

4. Otherwise, invokes `wakeup_softirqd()` to wake up, if necessary, the *ksoftirqd kernel thread of the local CPU (see later)*.

5. Executes the `local_irq_restore` macro to restore the state of the IF flag saved in step 1.

Checks for active (pending) softirqs should be perfomed periodically, but without inducing too much overhead. They are performed in a few points of the kernel code. Here is a list of the most significant points (be warned that number and position of the softirq checkpoints change both with the kernel version and with the supported hardware architecture):

- When the kernel invokes the `local_bh_enable()` function[*] to enable softirqs on the local CPU

- When the `do_IRQ()` function finishes handling an I/O interrupt and invokes the `irq_exit()` macro

- If the system uses an I/O APIC, when the `smp_apic_timer_interrupt()` function finishes handling a local timer interrupt (see the section "Timekeeping Architecture in Multiprocessor Systems" in Chapter 6)

- In multiprocessor systems, when a CPU finishes handling a function triggered by a `CALL_FUNCTION_VECTOR` interprocessor interrupt

- When one of the special *ksoftirqd/n* kernel threads is awakened (see later)

The do_softirq() function

If pending softirqs are detected at one such checkpoint (`local_softirq_pending()` is not zero), the kernel invokes `do_softirq()` to take care of them. This function performs the following actions:

1. If `in_interrupt()` yields the value one, this function returns. This situation indicates either that `do_softirq()` has been invoked in interrupt context or that the softirqs are currently disabled.

2. Executes `local_irq_save` to save the state of the IF flag and to disable the interrupts on the local CPU.

3. If the size of the `thread_union` structure is 4 KB, it switches to the soft IRQ stack, if necessary. This step is very similar to step 2 of `do_IRQ()` in the earlier section "I/O Interrupt Handling;" of course, the `softirq_ctx` array is used instead of `hardirq_ctx`.

4. Invokes the `__do_softirq()` function (see the following section).

---

[*] The name `local_bh_enable()` refers to a special type of deferrable function called "bottom half" that no longer exists in Linux 2.6.

5. If the soft IRQ stack has been effectively switched in step 3 above, it restores the original stack pointer into the `esp` register, thus switching back to the exception stack that was in use before.

6. Executes `local_irq_restore` to restore the state of the IF flag (local interrupts enabled or disabled) saved in step 2 and returns.

The __do_softirq() function

The `__do_softirq()` function reads the softirq bit mask of the local CPU and executes the deferrable functions corresponding to every set bit. While executing a softirq function, new pending softirqs might pop up; in order to ensure a low latency time for the deferrable funtions, `__do_softirq()` keeps running until all pending softirqs have been executed. This mechanism, however, could force `__do_softirq()` to run for long periods of time, thus considerably delaying User Mode processes. For that reason, `__do_softirq()` performs a fixed number of iterations and then returns. The remaining pending softirqs, if any, will be handled in due time by the *ksoftirqd* kernel thread described in the next section. Here is a short description of the actions performed by the function:

1. Initializes the iteration counter to 10.

2. Copies the softirq bit mask of the local CPU (selected by `local_softirq_pending()`) in the `pending` local variable.

3. Invokes `local_bh_disable()` to increase the softirq counter. It is somewhat counterintuitive that deferrable functions should be disabled before starting to execute them, but it really makes a lot of sense. Because the deferrable functions mostly run with interrupts enabled, an interrupt can be raised in the middle of the `__do_softirq()` function. When `do_IRQ()` executes the `irq_exit()` macro, another instance of the `__do_softirq()` function could be started. This has to be avoided, because deferrable functions must execute serially on the CPU. Thus, the first instance of `__do_softirq()` disables deferrable functions, so that every new instance of the function will exit at step 1 of `do_softirq()`.

4. Clears the softirq bitmap of the local CPU, so that new softirqs can be activated (the value of the bit mask has already been saved in the `pending` local variable in step 2).

5. Executes `local_irq_enable()` to enable local interrupts.

6. For each bit set in the `pending` local variable, it executes the corresponding softirq function; recall that the function address for the softirq with index n is stored in `softirq_vec[n]->action`.

7. Executes `local_irq_disable()` to disable local interrupts.

8. Copies the softirq bit mask of the local CPU into the `pending` local variable and decreases the iteration counter one more time.

9. If `pending` is not zero—at least one softirq has been activated since the start of the last iteration—and the iteration counter is still positive, it jumps back to step 4.

10. If there are more pending softirqs, it invokes `wakeup_softirqd()` to wake up the kernel thread that takes care of the softirqs for the local CPU (see next section).

11. Subtracts 1 from the softirq counter, thus reenabling the deferrable functions.

The ksoftirqd kernel threads

In recent kernel versions, each CPU has its own *ksoftirqd/n* kernel thread (where *n* is the logical number of the CPU). Each *ksoftirqd/n* kernel thread runs the `ksoftirqd()` function, which essentially executes the following loop:

```
for(;;) {
    set_current_state(TASK_INTERRUPTIBLE);
    schedule();
    /* now in TASK_RUNNING state */
    while (local_softirq_pending()) {
        preempt_disable();
        do_softirq();
        preempt_enable();
        cond_resched();
    }
}
```

When awakened, the kernel thread checks the `local_softirq_pending()` softirq bit mask and invokes, if necessary, `do_softirq()`. If there are no softirqs pending, the function puts the current process in the `TASK_INTERRUPTIBLE` state and invokes then the `cond_resched()` function to perform a process switch if required by the current process (flag `TIF_NEED_RESCHED` of the current `thread_info` set).

The *ksoftirqd/n* kernel threads represent a solution for a critical trade-off problem.

Softirq functions may reactivate themselves; in fact, both the networking softirqs and the tasklet softirqs do this. Moreover, external events, such as packet flooding on a network card, may activate softirqs at very high frequency.

The potential for a continuous high-volume flow of softirqs creates a problem that is solved by introducing kernel threads. Without them, developers are essentially faced with two alternative strategies.

The first strategy consists of ignoring new softirqs that occur while `do_softirq()` is running. In other words, the `do_softirq()` function could determine what softirqs are pending when the function is started and then execute their functions. Next, it would terminate without rechecking the pending softirqs. This solution is not good enough. Suppose that a softirq function is reactivated during the execution of `do_softirq()`. In the worst case, the softirq is not executed again until the next timer interrupt, even if the machine is idle. As a result, softirq latency time is unacceptable for networking developers.

The second strategy consists of continuously rechecking for pending softirqs. The `do_softirq()` function could keep checking the pending softirqs and would terminate only when none of them is pending. While this solution might satisfy networking developers, it can certainly upset normal users of the system: if a high-frequency flow of packets is received by a network card or a softirq function keeps activating itself, the `do_softirq()` function never returns, and the User Mode programs are virtually stopped.

The *ksoftirqd/n* kernel threads try to solve this difficult trade-off problem. The `do_softirq()` function determines what softirqs are pending and executes their functions. After a few iterations, if the flow of softirqs does not stop, the function wakes up the kernel thread and terminates (step 10 of `__do_softirq()`). The kernel thread has low priority, so user programs have a chance to run; but if the machine is idle, the pending softirqs are executed quickly.

Tasklets

Tasklets are the preferred way to implement deferrable functions in I/O drivers. As already explained, tasklets are built on top of two softirqs named `HI_SOFTIRQ` and `TASKLET_SOFTIRQ`. Several tasklets may be associated with the same softirq, each tasklet carrying its own function. There is no real difference between the two softirqs, except that `do_softirq()` executes `HI_SOFTIRQ`'s tasklets before `TASKLET_SOFTIRQ`'s tasklets.

Tasklets and high-priority tasklets are stored in the `tasklet_vec` and `tasklet_hi_vec` arrays, respectively. Both of them include `NR_CPUS` elements of type `tasklet_head`, and each element consists of a pointer to a list of *tasklet descriptors*. The tasklet descriptor is a data structure of type `tasklet_struct`, whose fields are shown in Table 4-11.

Table 4-11. The fields of the tasklet descriptor

Field name	Description
next	Pointer to next descriptor in the list
state	Status of the tasklet
count	Lock counter
func	Pointer to the tasklet function
data	An unsigned long integer that may be used by the tasklet function

The `state` field of the tasklet descriptor includes two flags:

TASKLET_STATE_SCHED
> When set, this indicates that the tasklet is pending (has been scheduled for execution); it also means that the tasklet descriptor is inserted in one of the lists of the `tasklet_vec` and `tasklet_hi_vec` arrays.

`TASKLET_STATE_RUN`

> When set, this indicates that the tasklet is being executed; on a uniprocessor system this flag is not used because there is no need to check whether a specific tasklet is running.

Let's suppose you're writing a device driver and you want to use a tasklet: what has to be done? First of all, you should allocate a new `tasklet_struct` data structure and initialize it by invoking `tasklet_init()`; this function receives as its parameters the address of the tasklet descriptor, the address of your tasklet function, and its optional integer argument.

The tasklet may be selectively disabled by invoking either `tasklet_disable_nosync()` or `tasklet_disable()`. Both functions increase the count field of the tasklet descriptor, but the latter function does not return until an already running instance of the tasklet function has terminated. To reenable the tasklet, use `tasklet_enable()`.

To activate the tasklet, you should invoke either the `tasklet_schedule()` function or the `tasklet_hi_schedule()` function, according to the priority that you require for the tasklet. The two functions are very similar; each of them performs the following actions:

1. Checks the `TASKLET_STATE_SCHED` flag; if it is set, returns (the tasklet has already been scheduled).
2. Invokes `local_irq_save` to save the state of the IF flag and to disable local interrupts.
3. Adds the tasklet descriptor at the beginning of the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]`, where `n` denotes the logical number of the local CPU.
4. Invokes `raise_softirq_irqoff()` to activate either the `TASKLET_SOFTIRQ` or the `HI_SOFTIRQ` softirq (this function is similar to `raise_softirq()`, except that it assumes that local interrupts are already disabled).
5. Invokes `local_irq_restore` to restore the state of the IF flag.

Finally, let's see how the tasklet is executed. We know from the previous section that, once activated, softirq functions are executed by the `do_softirq()` function. The softirq function associated with the `HI_SOFTIRQ` softirq is named `tasklet_hi_action()`, while the function associated with `TASKLET_SOFTIRQ` is named `tasklet_action()`. Once again, the two functions are very similar; each of them:

1. Disables local interrupts.
2. Gets the logical number `n` of the local CPU.
3. Stores the address of the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]` in the `list` local variable.

4. Puts a `NULL` address in `tasklet_vec[n]` or `tasklet_hi_vec[n]`, thus emptying the list of scheduled tasklet descriptors.

5. Enables local interrupts.

6. For each tasklet descriptor in the list pointed to by `list`:

   a. In multiprocessor systems, checks the `TASKLET_STATE_RUN` flag of the tasklet.

      - If it is set, a tasklet of the same type is already running on another CPU, so the function reinserts the task descriptor in the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]` and activates the `TASKLET_SOFTIRQ` or `HI_SOFTIRQ` softirq again. In this way, execution of the tasklet is deferred until no other tasklets of the same type are running on other CPUs.

      - Otherwise, the tasklet is not running on another CPU: sets the flag so that the tasklet function cannot be executed on other CPUs.

   b. Checks whether the tasklet is disabled by looking at the `count` field of the tasklet descriptor. If the tasklet is disabled, it clears its `TASKLET_STATE_RUN` flag and reinserts the task descriptor in the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]`; then the function activates the `TASKLET_SOFTIRQ` or `HI_SOFTIRQ` softirq again.

   c. If the tasklet is enabled, it clears the `TASKLET_STATE_SCHED` flag and executes the tasklet function.

Notice that, unless the tasklet function reactivates itself, every tasklet activation triggers at most one execution of the tasklet function.

Work Queues

The *work queues* have been introduced in Linux 2.6 and replace a similar construct called "task queue" used in Linux 2.4. They allow kernel functions to be activated (much like deferrable functions) and later executed by special kernel threads called *worker threads*.

Despite their similarities, deferrable functions and work queues are quite different. The main difference is that deferrable functions run in interrupt context while functions in work queues run in process context. Running in process context is the only way to execute functions that can block (for instance, functions that need to access some block of data on disk) because, as already observed in the section "Nested Execution of Exception and Interrupt Handlers" earlier in this chapter, no process switch can take place in interrupt context. Neither deferrable functions nor functions in a work queue can access the User Mode address space of a process. In fact, a deferrable function cannot make any assumption about the process that is currently running when it is executed. On the other hand, a function in a work queue is executed by a kernel thread, so there is no User Mode address space to access.

Work queue data structures

The main data structure associated with a work queue is a descriptor called workqueue_struct, which contains, among other things, an array of NR_CPUS elements, the maximum number of CPUs in the system.[*] Each element is a descriptor of type cpu_workqueue_struct, whose fields are shown in Table 4-12.

Table 4-12. The fields of the cpu_workqueue_struct structure

Field name	Description
lock	Spin lock used to protect the structure
remove_sequence	Sequence number used by flush_workqueue()
insert_sequence	Sequence number used by flush_workqueue()
worklist	Head of the list of pending functions
more_work	Wait queue where the worker thread waiting for more work to be done sleeps
work_done	Wait queue where the processes waiting for the work queue to be flushed sleep
wq	Pointer to the workqueue_struct structure containing this descriptor
thread	Process descriptor pointer of the worker thread of the structure
run_depth	Current execution depth of run_workqueue() (this field may become greater than one when a function in the work queue list blocks)

The worklist field of the cpu_workqueue_struct structure is the head of a doubly linked list collecting the pending functions of the work queue. Every pending function is represented by a work_struct data structure, whose fields are shown in Table 4-13.

Table 4-13. The fields of the work_struct structure

Field name	Description
pending	Set to 1 if the function is already in a work queue list, 0 otherwise
entry	Pointers to next and previous elements in the list of pending functions
func	Address of the pending function
data	Pointer passed as a parameter to the pending function
wq_data	Usually points to the parent cpu_workqueue_struct descriptor
timer	Software timer used to delay the execution of the pending function

Work queue functions

The create_workqueue("foo") function receives as its parameter a string of characters and returns the address of a workqueue_struct descriptor for the newly created

---

[*] The reason for duplicating the work queue data structures in multiprocessor systems is that per-CPU local data structures yield a much more efficient code (see the section "Per-CPU Variables" in Chapter 5).

work queue. The function also creates *n* worker threads (where *n* is the number of CPUs effectively present in the system), named after the string passed to the function: *foo/0*, *foo/1*, and so on. The create_singlethread_workqueue() function is similar, but it creates just one worker thread, no matter what the number of CPUs in the system is. To destroy a work queue the kernel invokes the destroy_workqueue() function, which receives as its parameter a pointer to a workqueue_struct array.

queue_work() inserts a function (already packaged inside a work_struct descriptor) in a work queue; it receives a pointer wq to the workqueue_struct descriptor and a pointer work to the work_struct descriptor. queue_work() essentially performs the following steps:

1. Checks whether the function to be inserted is already present in the work queue (work->pending field equal to 1); if so, terminates.
2. Adds the work_struct descriptor to the work queue list, and sets work->pending to 1.
3. If a worker thread is sleeping in the more_work wait queue of the local CPU's cpu_workqueue_struct descriptor, the function wakes it up.

The queue_delayed_work() function is nearly identical to queue_work(), except that it receives a third parameter representing a time delay in system ticks (see Chapter 6). It is used to ensure a minimum delay before the execution of the pending function. In practice, queue_delayed_work() relies on the software timer in the timer field of the work_struct descriptor to defer the actual insertion of the work_struct descriptor in the work queue list. cancel_delayed_work() cancels a previously scheduled work queue function, provided that the corresponding work_struct descriptor has not already been inserted in the work queue list.

Every worker thread continuously executes a loop inside the worker_thread() function; most of the time the thread is sleeping and waiting for some work to be queued. Once awakened, the worker thread invokes the run_workqueue() function, which essentially removes every work_struct descriptor from the work queue list of the worker thread and executes the corresponding pending function. Because work queue functions can block, the worker thread can be put to sleep and even migrated to another CPU when resumed.[*]

Sometimes the kernel has to wait until all pending functions in a work queue have been executed. The flush_workqueue() function receives a workqueue_struct descriptor address and blocks the calling process until all functions that are pending in the work queue terminate. The function, however, does not wait for any pending function that was added to the work queue following flush_workqueue() invocation; the

---

[*] Strangely enough, a worker thread can be executed by every CPU, not just the CPU corresponding to the cpu_workqueue_struct descriptor to which the worker thread belongs. Therefore, queue_work() inserts a function in the queue of the local CPU, but that function may be executed by any CPU in the systems.

remove_sequence and insert_sequence fields of every cpu_workqueue_struct descriptor are used to recognize the newly added pending functions.

The predefined work queue

In most cases, creating a whole set of worker threads in order to run a function is overkill. Therefore, the kernel offers a predefined work queue called *events*, which can be freely used by every kernel developer. The predefined work queue is nothing more than a standard work queue that may include functions of different kernel layers and I/O drivers; its workqueue_struct descriptor is stored in the keventd_wq array. To make use of the predefined work queue, the kernel offers the functions listed in Table 4-14.

Table 4-14. Helper functions for the predefined work queue

Predefined work queue function	Equivalent standard work queue function
schedule_work(w)	queue_work(keventd_wq,w)
schedule_delayed_work(w,d)	queue_delayed_work(keventd_wq,w,d) (on any CPU)
schedule_delayed_work_on(cpu,w,d)	queue_delayed_work(keventd_wq,w,d) (on a given CPU)
flush_scheduled_work()	flush_workqueue(keventd_wq)

The predefined work queue saves significant system resources when the function is seldom invoked. On the other hand, functions executed in the predefined work queue should not block for a long time: because the execution of the pending functions in the work queue list is serialized on each CPU, a long delay negatively affects the other users of the predefined work queue.

In addition to the general *events* queue, you'll find a few specialized work queues in Linux 2.6. The most significant is the *kblockd* work queue used by the block device layer (see Chapter 14).

Returning from Interrupts and Exceptions

We will finish the chapter by examining the termination phase of interrupt and exception handlers. (Returning from a system call is a special case, and we shall describe it in Chapter 10.) Although the main objective is clear—namely, to resume execution of some program—several issues must be considered before doing it:

Number of kernel control paths being concurrently executed
> If there is just one, the CPU must switch back to User Mode.

Pending process switch requests
> If there is any request, the kernel must perform process scheduling; otherwise, control is returned to the current process.

Pending signals
> If a signal is sent to the current process, it must be handled.

Single-step mode
> If a debugger is tracing the execution of the current process, single-step mode must be restored before switching back to User Mode.

Virtual-8086 mode
> If the CPU is in virtual-8086 mode, the current process is executing a legacy Real Mode program, thus it must be handled in a special way.

A few flags are used to keep track of pending process switch requests, of pending signals, and of single step execution; they are stored in the `flags` field of the `thread_info` descriptor. The field stores other flags as well, but they are not related to returning from interrupts and exceptions. See Table 4-15 for a complete list of these flags.

Table 4-15. The flags field of the thread_info descriptor (continues)

Flag name	Description
TIF_SYSCALL_TRACE	System calls are being traced
TIF_NOTIFY_RESUME	Not used in the 80×86 platform
TIF_SIGPENDING	The process has pending signals
TIF_NEED_RESCHED	Scheduling must be performed
TIF_SINGLESTEP	Restore single step execution on return to User Mode
TIF_IRET	Force return from system call via `iret` rather than `sysexit`
TIF_SYSCALL_AUDIT	System calls are being audited
TIF_POLLING_NRFLAG	The idle process is polling the `TIF_NEED_RESCHED` flag
TIF_MEMDIE	The process is being destroyed to reclaim memory (see the section "The Out of Memory Killer" in Chapter 17)

The kernel assembly language code that accomplishes all these things is not, technically speaking, a function, because control is never returned to the functions that invoke it. It is a piece of code with two different entry points: `ret_from_intr()` and `ret_from_exception()`. As their names suggest, the kernel enters the former when terminating an interrupt handler, and it enters the latter when terminating an exception handler. We shall refer to the two entry points as functions, because this makes the description simpler.

The general flow diagram with the corresponding two entry points is illustrated in Figure 4-6. The gray boxes refer to assembly language instructions that implement kernel preemption (see Chapter 5); if you want to see what the kernel does when it is compiled without support for kernel preemption, just ignore the gray boxes. The `ret_from_exception()` and `ret_from_intr()` entry points look quite similar in the flow diagram. A difference exists only if support for kernel preemption has been selected as a compilation option: in this case, local interrupts are immediately disabled when returning from exceptions.

Figure 4-6. Returning from interrupts and exceptions

The flow diagram gives a rough idea of the steps required to resume the execution of an interrupted program. Now we will go into detail by discussing the assembly language code.

The entry points

The ret_from_intr() and ret_from_exception() entry points are essentially equivalent to the following assembly language code:

```
ret_from_exception:
    cli ; missing if kernel preemption is not supported
ret_from_intr:
    movl $-8192, %ebp ; -4096 if multiple Kernel Mode stacks are used
    andl %esp, %ebp
```

```
        movl 0x30(%esp), %eax
        movb 0x2c(%esp), %al
        testl $0x00020003, %eax
        jnz resume_userspace
        jpm resume_kernel
```

Recall that when returning from an interrupt, the local interrupts are disabled (see step 3 in the earlier description of handle_IRQ_event()); thus, the cli assembly language instruction is executed only when returning from an exception.

The kernel loads the address of the thread_info descriptor of current in the ebp register (see "Identifying a Process" in Chapter 3).

Next, the values of the cs and eflags registers, which were pushed on the stack when the interrupt or the exception occurred, are used to determine whether the interrupted program was running in User Mode, or if the VM flag of eflags was set.[*] In either case, a jump is made to the resume_userspace label. Otherwise, a jump is made to the resume_kernel label.

Resuming a kernel control path

The assembly language code at the resume_kernel label is executed if the program to be resumed is running in Kernel Mode:

```
resume_kernel:
    cli                 ; these three instructions are
    cmpl $0, 0x14(%ebp) ; missing if kernel preemption
    jz need_resched     ; is not supported
restore_all:
    popl %ebx
    popl %ecx
    popl %edx
    popl %esi
    popl %edi
    popl %ebp
    popl %eax
    popl %ds
    popl %es
    addl $4, %esp
    iret
```

If the preempt_count field of the thread_info descriptor is zero (kernel preemption enabled), the kernel jumps to the need_resched label. Otherwise, the interrupted program is to be restarted. The function loads the registers with the values saved when the interrupt or the exception started, and the function yields control by executing the iret instruction.

---

[*] When this flag is set, programs are executed in virtual-8086 mode; see the Pentium manuals for more details.

Checking for kernel preemption

When this code is executed, none of the unfinished kernel control paths is an interrupt handler, otherwise the preempt_count field would be greater than zero. However, as stated in "Nested Execution of Exception and Interrupt Handlers" earlier in this chapter, there could be up to two kernel control paths associated with exceptions (beside the one that is terminating).

```
need_resched:
    movl 0x8(%ebp), %ecx
    testb $(1<<TIF_NEED_RESCHED), %cl
    jz restore_all
    testl $0x00000200,0x30(%esp)
    jz restore_all
    call preempt_schedule_irq
    jmp need_resched
```

If the TIF_NEED_RESCHED flag in the flags field of current->thread_info is zero, no process switch is required, thus a jump is made to the restore_all label. Also a jump to the same label is made if the kernel control path that is being resumed was running with the local interrupts disabled. In this case a process switch could corrupt kernel data structures (see the section "When Synchronization Is Necessary" in Chapter 5 for more details).

If a process switch is required, the preempt_schedule_irq() function is invoked: it sets the PREEMPT_ACTIVE flag in the preempt_count field, temporarily sets the big kernel lock counter to -1 (see the section "The Big Kernel Lock" in Chapter 5), enables the local interrupts, and invokes schedule() to select another process to run. When the former process will resume, preempt_schedule_irq() restores the previous value of the big kernel lock counter, clears the PREEMPT_ACTIVE flag, and disables local interrupts. The schedule() function will continue to be invoked as long as the TIF_NEED_RESCHED flag of the current process is set.

Resuming a User Mode program

If the program to be resumed was running in User Mode, a jump is made to the resume_userspace label:

```
resume_userspace:
    cli
    movl 0x8(%ebp), %ecx
    andl $0x0000ff6e, %ecx
    je restore_all
    jmp work_pending
```

After disabling the local interrupts, a check is made on the value of the flags field of current->thread_info. If no flag except TIF_SYSCALL_TRACE, TIF_SYSCALL_AUDIT, or TIF_SINGLESTEP is set, nothing remains to be done: a jump is made to the restore_all label, thus resuming the User Mode program.

Checking for rescheduling

The flags in the thread_info descriptor state that additional work is required before resuming the interrupted program.

```
work_pending:
    testb $(1<<TIF_NEED_RESCHED), %cl
    jz work_notifysig
work_resched:
    call schedule
    cli
    jmp resume_userspace
```

If a process switch request is pending, schedule() is invoked to select another process to run. When the former process will resume, a jump is made back to resume_userspace.

Handling pending signals, virtual-8086 mode, and single stepping

There is other work to be done besides process switch requests:

```
work_notifysig:
    movl %esp, %eax
    testl $0x00020000, 0x30(%esp)
    je 1f
work_notifysig_v86:
    pushl %ecx
    call save_v86_state
    popl %ecx
    movl %eax, %esp
1:
    xorl %edx, %edx
    call do_notify_resume
    jmp restore_all
```

If the VM control flag in the eflags register of the User Mode program is set, the save_v86_state() function is invoked to build up the virtual-8086 mode data structures in the User Mode address space. Then the do_notify_resume() function is invoked to take care of pending signals and single stepping. Finally, a jump is made to the restore_all label to resume the interrupted program.

Kernel Synchronization

You could think of the kernel as a server that answers requests; these requests can come either from a process running on a CPU or an external device issuing an interrupt request. We make this analogy to underscore that parts of the kernel are not run serially, but in an interleaved way. Thus, they can give rise to race conditions, which must be controlled through proper synchronization techniques. A general introduction to these topics can be found in the section "An Overview of Unix Kernels" in Chapter 1.

We start this chapter by reviewing when, and to what extent, kernel requests are executed in an interleaved fashion. We then introduce the basic synchronization primitives implemented by the kernel and describe how they are applied in the most common cases. Finally, we illustrate a few practical examples.

How the Kernel Services Requests

To get a better grasp of how kernel's code is executed, we will look at the kernel as a waiter who must satisfy two types of requests: those issued by customers and those issued by a limited number of different bosses. The policy adopted by the waiter is the following:

1. If a boss calls while the waiter is idle, the waiter starts servicing the boss.

2. If a boss calls while the waiter is servicing a customer, the waiter stops servicing the customer and starts servicing the boss.

3. If a boss calls while the waiter is servicing another boss, the waiter stops servicing the first boss and starts servicing the second one. When he finishes servicing the new boss, he resumes servicing the former one.

4. One of the bosses may induce the waiter to leave the customer being currently serviced. After servicing the last request of the bosses, the waiter may decide to drop temporarily his customer and to pick up a new one.

The services performed by the waiter correspond to the code executed when the CPU is in Kernel Mode. If the CPU is executing in User Mode, the waiter is considered idle.

Boss requests correspond to interrupts, while customer requests correspond to system calls or exceptions raised by User Mode processes. As we shall see in detail in Chapter 10, User Mode processes that want to request a service from the kernel must issue an appropriate instruction (on the 80×86, an `int $0x80` or a `sysenter` instruction). Such instructions raise an exception that forces the CPU to switch from User Mode to Kernel Mode. In the rest of this chapter, we will generally denote as "exceptions" both the system calls and the usual exceptions.

The careful reader has already associated the first three rules with the nesting of kernel control paths described in "Nested Execution of Exception and Interrupt Handlers" in Chapter 4. The fourth rule corresponds to one of the most interesting new features included in the Linux 2.6 kernel, namely *kernel preemption*.

Kernel Preemption

It is surprisingly hard to give a good definition of kernel preemption. As a first try, we could say that a kernel is *preemptive* if a process switch may occur while the replaced process is executing a kernel function, that is, while it runs in Kernel Mode. Unfortunately, in Linux (as well as in any other real operating system) things are much more complicated:

- Both in preemptive and nonpreemptive kernels, a process running in Kernel Mode can voluntarily relinquish the CPU, for instance because it has to sleep waiting for some resource. We will call this kind of process switch a *planned process switch*. However, a preemptive kernel differs from a nonpreemptive kernel on the way a process running in Kernel Mode reacts to asynchronous events that could induce a process switch—for instance, an interrupt handler that awakes a higher priority process. We will call this kind of process switch a *forced process switch*.

- All process switches are performed by the `switch_to` macro. In both preemptive and nonpreemptive kernels, a process switch occurs when a process has finished some thread of kernel activity and the scheduler is invoked. However, in nonpreemptive kernels, the current process cannot be replaced unless it is about to switch to User Mode (see the section "Performing the Process Switch" in Chapter 3).

Therefore, the main characteristic of a preemptive kernel is that a process running in Kernel Mode can be replaced by another process while in the middle of a kernel function.

Let's give a couple of examples to illustrate the difference between preemptive and nonpreemptive kernels.

While process A executes an exception handler (necessarily in Kernel Mode), a higher priority process B becomes runnable. This could happen, for instance, if an IRQ occurs

and the corresponding handler awakens process B. If the kernel is preemptive, a forced process switch replaces process A with B. The exception handler is left unfinished and will be resumed only when the scheduler selects again process A for execution. Conversely, if the kernel is nonpreemptive, no process switch occurs until process A either finishes handling the exception handler or voluntarily relinquishes the CPU.

For another example, consider a process that executes an exception handler and whose time quantum expires (see the section "The scheduler_tick( ) Function" in Chapter 7). If the kernel is preemptive, the process may be replaced immediately; however, if the kernel is nonpreemptive, the process continues to run until it finishes handling the exception handler or voluntarily relinquishes the CPU.

The main motivation for making a kernel preemptive is to reduce the *dispatch latency* of the User Mode processes, that is, the delay between the time they become runnable and the time they actually begin running. Processes performing timely scheduled tasks (such as external hardware controllers, environmental monitors, movie players, and so on) really benefit from kernel preemption, because it reduces the risk of being delayed by another process running in Kernel Mode.

Making the Linux 2.6 kernel preemptive did not require a drastic change in the kernel design with respect to the older nonpreemptive kernel versions. As described in the section "Returning from Interrupts and Exceptions" in Chapter 4, kernel preemption is disabled when the `preempt_count` field in the `thread_info` descriptor referenced by the `current_thread_info( )` macro is greater than zero. The field encodes three different counters, as shown in Table 4-10 in Chapter 4, so it is greater than zero when any of the following cases occurs:

1. The kernel is executing an interrupt service routine.
2. The deferrable functions are disabled (always true when the kernel is executing a softirq or tasklet).
3. The kernel preemption has been explicitly disabled by setting the preemption counter to a positive value.

The above rules tell us that the kernel can be preempted only when it is executing an exception handler (in particular a system call) and the kernel preemption has not been explicitly disabled. Furthermore, as described in the section "Returning from Interrupts and Exceptions" in Chapter 4, the local CPU must have local interrupts enabled, otherwise kernel preemption is not performed.

A few simple macros listed in Table 5-1 deal with the preemption counter in the `preempt_count` field.

Table 5-1. Macros dealing with the preemption counter subfield

Macro	Description
preempt_count( )	Selects the `preempt_count` field in the `thread_info` descriptor
preempt_disable( )	Increases by one the value of the preemption counter

Table 5-1. Macros dealing with the preemption counter subfield (continued)

Macro	Description
`preempt_enable_no_resched()`	Decreases by one the value of the preemption counter
`preempt_enable()`	Decreases by one the value of the preemption counter, and invokes `preempt_schedule()` if the `TIF_NEED_RESCHED` flag in the `thread_info` descriptor is set
`get_cpu()`	Similar to `preempt_disable()`, but also returns the number of the local CPU
`put_cpu()`	Same as `preempt_enable()`
`put_cpu_no_resched()`	Same as `preempt_enable_no_resched()`

The `preempt_enable()` macro decreases the preemption counter, then checks whether the `TIF_NEED_RESCHED` flag is set (see Table 4-15 in Chapter 4). In this case, a process switch request is pending, so the macro invokes the `preempt_schedule()` function, which essentially executes the following code:

```
if (!current_thread_info->preempt_count && !irqs_disabled()) {
    current_thread_info->preempt_count = PREEMPT_ACTIVE;
    schedule();
    current_thread_info->preempt_count = 0;
}
```

The function checks whether local interrupts are enabled and the `preempt_count` field of `current` is zero; if both conditions are true, it invokes `schedule()` to select another process to run. Therefore, kernel preemption may happen either when a kernel control path (usually, an interrupt handler) is terminated, or when an exception handler reenables kernel preemption by means of `preempt_enable()`. As we'll see in the section "Disabling and Enabling Deferrable Functions" later in this chapter, kernel preemption may also happen when deferrable functions are enabled.

We'll conclude this section by noticing that kernel preemption introduces a nonnegligible overhead. For that reason, Linux 2.6 features a kernel configuration option that allows users to enable or disable kernel preemption when compiling the kernel.

When Synchronization Is Necessary

Chapter 1 introduced the concepts of race condition and critical region for processes. The same definitions apply to kernel control paths. In this chapter, a race condition can occur when the outcome of a computation depends on how two or more interleaved kernel control paths are nested. A *critical region* is a section of code that must be completely executed by the kernel control path that enters it before another kernel control path can enter it.

Interleaving kernel control paths complicates the life of kernel developers: they must apply special care in order to identify the critical regions in exception handlers, interrupt handlers, deferrable functions, and kernel threads. Once a critical region has

been identified, it must be suitably protected to ensure that any time at most one kernel control path is inside that region.

Suppose, for instance, that two different interrupt handlers need to access the same data structure that contains several related member variables—for instance, a buffer and an integer indicating its length. All statements affecting the data structure must be put into a single critical region. If the system includes a single CPU, the critical region can be implemented by disabling interrupts while accessing the shared data structure, because nesting of kernel control paths can only occur when interrupts are enabled.

On the other hand, if the same data structure is accessed only by the service routines of system calls, and if the system includes a single CPU, the critical region can be implemented quite simply by disabling kernel preemption while accessing the shared data structure.

As you would expect, things are more complicated in multiprocessor systems. Many CPUs may execute kernel code at the same time, so kernel developers cannot assume that a data structure can be safely accessed just because kernel preemption is disabled and the data structure is never addressed by an interrupt, exception, or softirq handler.

We'll see in the following sections that the kernel offers a wide range of different synchronization techniques. It is up to kernel designers to solve each synchronization problem by selecting the most efficient technique.

When Synchronization Is Not Necessary

Some design choices already discussed in the previous chapter simplify somewhat the synchronization of kernel control paths. Let us recall them briefly:

- All interrupt handlers acknowledge the interrupt on the PIC and also disable the IRQ line. Further occurrences of the same interrupt cannot occur until the handler terminates.

- Interrupt handlers, softirqs, and tasklets are both nonpreemptable and nonblocking, so they cannot be suspended for a long time interval. In the worst case, their execution will be slightly delayed, because other interrupts occur during their execution (nested execution of kernel control paths).

- A kernel control path performing interrupt handling cannot be interrupted by a kernel control path executing a deferrable function or a system call service routine.

- Softirqs and tasklets cannot be interleaved on a given CPU.

- The same tasklet cannot be executed simultaneously on several CPUs.

Each of the above design choices can be viewed as a constraint that can be exploited to code some kernel functions more easily. Here are a few examples of possible simplifications:

- Interrupt handlers and tasklets need not to be coded as reentrant functions.

- Per-CPU variables accessed by softirqs and tasklets only do not require synchronization.
- A data structure accessed by only one kind of tasklet does not require synchronization.

The rest of this chapter describes what to do when synchronization *is* necessary—i.e., how to prevent data corruption due to unsafe accesses to shared data structures.

Synchronization Primitives

We now examine how kernel control paths can be interleaved while avoiding race conditions among shared data. Table 5-2 lists the synchronization techniques used by the Linux kernel. The "Scope" column indicates whether the synchronization technique applies to all CPUs in the system or to a single CPU. For instance, local interrupt disabling applies to just one CPU (other CPUs in the system are not affected); conversely, an atomic operation affects all CPUs in the system (atomic operations on several CPUs cannot interleave while accessing the same data structure).

Table 5-2. Various types of synchronization techniques used by the kernel

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Let's now briefly discuss each synchronization technique. In the later section "Synchronizing Accesses to Kernel Data Structures," we show how these synchronization techniques can be combined to effectively protect kernel data structures.

Per-CPU Variables

The best synchronization technique consists in designing the kernel so as to avoid the need for synchronization in the first place. As we'll see, in fact, every explicit synchronization primitive has a significant performance cost.

The simplest and most efficient synchronization technique consists of declaring kernel variables as *per-CPU variables*. Basically, a per-CPU variable is an array of data structures, one element per each CPU in the system.

A CPU should not access the elements of the array corresponding to the other CPUs; on the other hand, it can freely read and modify its own element without fear of race conditions, because it is the only CPU entitled to do so. This also means, however, that the per-CPU variables can be used only in particular cases—basically, when it makes sense to logically split the data across the CPUs of the system.

The elements of the per-CPU array are aligned in main memory so that each data structure falls on a different line of the hardware cache (see the section "Hardware Cache" in Chapter 2). Therefore, concurrent accesses to the per-CPU array do not result in cache line snooping and invalidation, which are costly operations in terms of system performance.

While per-CPU variables provide protection against concurrent accesses from several CPUs, they do not provide protection against accesses from asynchronous functions (interrupt handlers and deferrable functions). In these cases, additional synchronization primitives are required.

Furthermore, per-CPU variables are prone to race conditions caused by kernel preemption, both in uniprocessor and multiprocessor systems. As a general rule, a kernel control path should access a per-CPU variable with kernel preemption disabled. Just consider, for instance, what would happen if a kernel control path gets the address of its local copy of a per-CPU variable, and then it is preempted and moved to another CPU: the address still refers to the element of the previous CPU.

Table 5-3 lists the main functions and macros offered by the kernel to use per-CPU variables.

Table 5-3. Functions and macros for the per-CPU variables

Macro or function name	Description
`DEFINE_PER_CPU(type, name)`	Statically allocates a per-CPU array called `name` of `type` data structures
`per_cpu(name, cpu)`	Selects the element for CPU `cpu` of the per-CPU array `name`
`__get_cpu_var(name)`	Selects the local CPU's element of the per-CPU array `name`
`get_cpu_var(name)`	Disables kernel preemption, then selects the local CPU's element of the per-CPU array `name`
`put_cpu_var(name)`	Enables kernel preemption (`name` is not used)
`alloc_percpu(type)`	Dynamically allocates a per-CPU array of `type` data structures and returns its address
`free_percpu(pointer)`	Releases a dynamically allocated per-CPU array at address `pointer`
`per_cpu_ptr(pointer, cpu)`	Returns the address of the element for CPU `cpu` of the per-CPU array at address `pointer`

Atomic Operations

Several assembly language instructions are of type "read-modify-write"—that is, they access a memory location twice, the first time to read the old value and the second time to write a new value.

Suppose that two kernel control paths running on two CPUs try to "read-modify-write" the same memory location at the same time by executing nonatomic operations. At first, both CPUs try to read the same location, but the *memory arbiter* (a hardware circuit that serializes accesses to the RAM chips) steps in to grant access to one of them and delay the other. However, when the first read operation has completed, the delayed CPU reads exactly the same (old) value from the memory location. Both CPUs then try to write the same (new) value to the memory location; again, the bus memory access is serialized by the memory arbiter, and eventually both write operations succeed. However, the global result is incorrect because both CPUs write the same (new) value. Thus, the two interleaving "read-modify-write" operations act as a single one.

The easiest way to prevent race conditions due to "read-modify-write" instructions is by ensuring that such operations are atomic at the chip level. Every such operation must be executed in a single instruction without being interrupted in the middle and avoiding accesses to the same memory location by other CPUs. These very small *atomic operations* can be found at the base of other, more flexible mechanisms to create critical regions.

Let's review 80×86 instructions according to that classification:

- Assembly language instructions that make zero or one aligned memory access are atomic.[*]
- Read-modify-write assembly language instructions (such as `inc` or `dec`) that read data from memory, update it, and write the updated value back to memory are atomic if no other processor has taken the memory bus after the read and before the write. Memory bus stealing never happens in a uniprocessor system.
- Read-modify-write assembly language instructions whose opcode is prefixed by the `lock` byte (`0xf0`) are atomic even on a multiprocessor system. When the control unit detects the prefix, it "locks" the memory bus until the instruction is finished. Therefore, other processors cannot access the memory location while the locked instruction is being executed.
- Assembly language instructions whose opcode is prefixed by a `rep` byte (`0xf2`, `0xf3`, which forces the control unit to repeat the same instruction several times) are not atomic. The control unit checks for pending interrupts before executing a new iteration.

When you write C code, you cannot guarantee that the compiler will use an atomic instruction for an operation like `a=a+1` or even for `a++`. Thus, the Linux kernel provides a special `atomic_t` type (an atomically accessible counter) and some special

---

[*] A data item is aligned in memory when its address is a multiple of its size in bytes. For instance, the address of an aligned short integer must be a multiple of two, while the address of an aligned integer must be a multiple of four. Generally speaking, an unaligned memory access is not atomic.

functions and macros (see Table 5-4) that act on `atomic_t` variables and are implemented as single, atomic assembly language instructions. On multiprocessor systems, each such instruction is prefixed by a `lock` byte.

Table 5-4. Atomic operations in Linux

Function	Description
atomic_read(v)	Return *v
atomic_set(v,i)	Set *v to i
atomic_add(i,v)	Add i to *v
atomic_sub(i,v)	Subtract i from *v
atomic_sub_and_test(i, v)	Subtract i from *v and return 1 if the result is zero; 0 otherwise
atomic_inc(v)	Add 1 to *v
atomic_dec(v)	Subtract 1 from *v
atomic_dec_and_test(v)	Subtract 1 from *v and return 1 if the result is zero; 0 otherwise
atomic_inc_and_test(v)	Add 1 to *v and return 1 if the result is zero; 0 otherwise
atomic_add_negative(i, v)	Add i to *v and return 1 if the result is negative; 0 otherwise
atomic_inc_return(v)	Add 1 to *v and return the new value of *v
atomic_dec_return(v)	Subtract 1 from *v and return the new value of *v
atomic_add_return(i, v)	Add i to *v and return the new value of *v
atomic_sub_return(i, v)	Subtract i from *v and return the new value of *v

Another class of atomic functions operate on bit masks (see Table 5-5).

Table 5-5. Atomic bit handling functions in Linux

Function	Description
test_bit(nr, addr)	Return the value of the nr^{th} bit of *addr
set_bit(nr, addr)	Set the nr^{th} bit of *addr
clear_bit(nr, addr)	Clear the nr^{th} bit of *addr
change_bit(nr, addr)	Invert the nr^{th} bit of *addr
test_and_set_bit(nr, addr)	Set the nr^{th} bit of *addr and return its old value
test_and_clear_bit(nr, addr)	Clear the nr^{th} bit of *addr and return its old value
test_and_change_bit(nr, addr)	Invert the nr^{th} bit of *addr and return its old value
atomic_clear_mask(mask, addr)	Clear all bits of *addr specified by mask
atomic_set_mask(mask, addr)	Set all bits of *addr specified by mask

Optimization and Memory Barriers

When using optimizing compilers, you should never take for granted that instructions will be performed in the exact order in which they appear in the source code. For example, a compiler might reorder the assembly language instructions in such a

way to optimize how registers are used. Moreover, modern CPUs usually execute several instructions in parallel and might reorder memory accesses. These kinds of reordering can greatly speed up the program.

When dealing with synchronization, however, reordering instructions must be avoided. Things would quickly become hairy if an instruction placed after a synchronization primitive is executed before the synchronization primitive itself. Therefore, all synchronization primitives act as optimization and memory barriers.

An *optimization barrier* primitive ensures that the assembly language instructions corresponding to C statements placed before the primitive are not mixed by the compiler with assembly language instructions corresponding to C statements placed after the primitive. In Linux the `barrier()` macro, which expands into `asm volatile("":::"memory")`, acts as an optimization barrier. The `asm` instruction tells the compiler to insert an assembly language fragment (empty, in this case). The `volatile` keyword forbids the compiler to reshuffle the `asm` instruction with the other instructions of the program. The `memory` keyword forces the compiler to assume that all memory locations in RAM have been changed by the assembly language instruction; therefore, the compiler cannot optimize the code by using the values of memory locations stored in CPU registers before the `asm` instruction. Notice that the optimization barrier does not ensure that the executions of the assembly language instructions are not mixed by the CPU— this is a job for a memory barrier.

A *memory barrier* primitive ensures that the operations placed before the primitive are finished before starting the operations placed after the primitive. Thus, a memory barrier is like a firewall that cannot be passed by an assembly language instruction.

In the 80×86 processors, the following kinds of assembly language instructions are said to be "serializing" because they act as memory barriers:

- All instructions that operate on I/O ports
- All instructions prefixed by the `lock` byte (see the section "Atomic Operations")
- All instructions that write into control registers, system registers, or debug registers (for instance, `cli` and `sti`, which change the status of the IF flag in the `eflags` register)
- The `lfence`, `sfence`, and `mfence` assembly language instructions, which have been introduced in the Pentium 4 microprocessor to efficiently implement read memory barriers, write memory barriers, and read-write memory barriers, respectively.
- A few special assembly language instructions; among them, the `iret` instruction that terminates an interrupt or exception handler

Linux uses a few memory barrier primitives, which are shown in Table 5-6. These primitives act also as optimization barriers, because we must make sure the compiler does not move the assembly language instructions around the barrier. "Read memory

barriers" act only on instructions that read from memory, while "write memory barriers" act only on instructions that write to memory. Memory barriers can be useful in both multiprocessor and uniprocessor systems. The `smp_xxx()` primitives are used whenever the memory barrier should prevent race conditions that might occur only in multiprocessor systems; in uniprocessor systems, they do nothing. The other memory barriers are used to prevent race conditions occurring both in uniprocessor and multiprocessor systems.

Table 5-6. Memory barriers in Linux

Macro	Description
mb( )	Memory barrier for MP and UP
rmb( )	Read memory barrier for MP and UP
wmb( )	Write memory barrier for MP and UP
smp_mb( )	Memory barrier for MP only
smp_rmb( )	Read memory barrier for MP only
smp_wmb( )	Write memory barrier for MP only

The implementations of the memory barrier primitives depend on the architecture of the system. On an 80×86 microprocessor, the `rmb()` macro usually expands into `asm volatile("lfence")` if the CPU supports the `lfence` assembly language instruction, or into `asm volatile("lock;addl $0,0(%%esp)":::"memory")` otherwise. The `asm` statement inserts an assembly language fragment in the code generated by the compiler and acts as an optimization barrier. The `lock; addl $0,0(%%esp)` assembly language instruction adds zero to the memory location on top of the stack; the instruction is useless by itself, but the `lock` prefix makes the instruction a memory barrier for the CPU.

The `wmb()` macro is actually simpler because it expands into `barrier()`. This is because existing Intel microprocessors never reorder write memory accesses, so there is no need to insert a serializing assembly language instruction in the code. The macro, however, forbids the compiler from shuffling the instructions.

Notice that in multiprocessor systems, all atomic operations described in the earlier section "Atomic Operations" act as memory barriers because they use the `lock` byte.

Spin Locks

A widely used synchronization technique is *locking*. When a kernel control path must access a shared data structure or enter a critical region, it needs to acquire a "lock" for it. A resource protected by a locking mechanism is quite similar to a resource confined in a room whose door is locked when someone is inside. If a kernel control path wishes to access the resource, it tries to "open the door" by acquiring the lock. It succeeds only if the resource is free. Then, as long as it wants to use the resource, the door remains locked. When the kernel control path releases the lock, the door is unlocked and another kernel control path may enter the room.

Figure 5-1 illustrates the use of locks. Five kernel control paths (P0, P1, P2, and P4) are trying to access two critical regions (C1 and C2). Kernel control path P0 is inside C1, while P2 and P4 are waiting to enter it. At the same time, P1 is inside C2, while P3 is waiting to enter it. Notice that P0 and P1 could run concurrently. The lock for critical region C3 is open because no kernel control path needs to enter it.
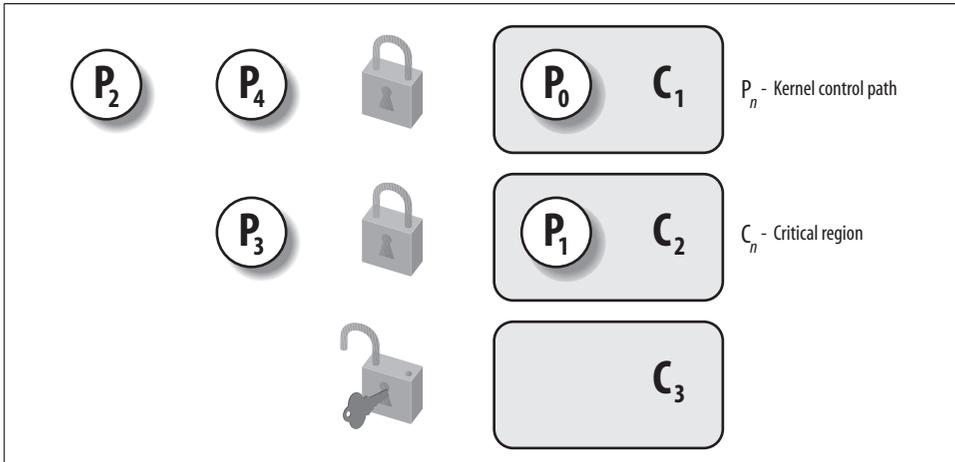
Figure 5-1. Protecting critical regions with several locks

Spin locks are a special kind of lock designed to work in a multiprocessor environment. If the kernel control path finds the spin lock "open," it acquires the lock and continues its execution. Conversely, if the kernel control path finds the lock "closed" by a kernel control path running on another CPU, it "spins" around, repeatedly executing a tight instruction loop, until the lock is released.

The instruction loop of spin locks represents a "busy wait." The waiting kernel control path keeps running on the CPU, even if it has nothing to do besides waste time. Nevertheless, spin locks are usually convenient, because many kernel resources are locked for a fraction of a millisecond only; therefore, it would be far more time-consuming to release the CPU and reacquire it later.

As a general rule, kernel preemption is disabled in every critical region protected by spin locks. In the case of a uniprocessor system, the locks themselves are useless, and the spin lock primitives just disable or enable the kernel preemption. Please notice that kernel preemption is still enabled during the busy wait phase, thus a process waiting for a spin lock to be released could be replaced by a higher priority process.

In Linux, each spin lock is represented by a `spinlock_t` structure consisting of two fields:

slock

    Encodes the spin lock state: the value 1 corresponds to the unlocked state, while every negative value and 0 denote the locked state

break_lock

    Flag signaling that a process is busy waiting for the lock (present only if the ker-
nel supports both SMP and kernel preemption)

Six macros shown in Table 5-7 are used to initialize, test, and set spin locks. All these
macros are based on atomic operations; this ensures that the spin lock will be
updated properly even when multiple processes running on different CPUs try to
modify the lock at the same time.[*]

Table 5-7. Spin lock macros

Macro	Description
spin_lock_init()	Set the spin lock to 1 (unlocked)
spin_lock()	Cycle until spin lock becomes 1 (unlocked), then set it to 0 (locked)
spin_unlock()	Set the spin lock to 1 (unlocked)
spin_unlock_wait()	Wait until the spin lock becomes 1 (unlocked)
spin_is_locked()	Return 0 if the spin lock is set to 1 (unlocked); 1 otherwise
spin_trylock()	Set the spin lock to 0 (locked), and return 1 if the previous value of the lock was 1; 0 oth-erwise

The spin_lock macro with kernel preemption

Let's discuss in detail the spin_lock macro, which is used to acquire a spin lock. The
following description refers to a preemptive kernel for an SMP system. The macro
takes the address slp of the spin lock as its parameter and executes the following
actions:

1. Invokes preempt_disable() to disable kernel preemption.

2. Invokes the _raw_spin_trylock() function, which does an atomic test-and-set
   operation on the spin lock's slock field; this function executes first some instruc-
   tions equivalent to the following assembly language fragment:

   ```
   movb $0, %al
   xchgb %al, slp->slock
   ```

   The xchg assembly language instruction exchanges atomically the content of the
   8-bit %al register (storing zero) with the content of the memory location pointed
   to by slp->slock. The function then returns the value 1 if the old value stored in
   the spin lock (in %al after the xchg instruction) was positive, the value 0 other-
   wise.

3. If the old value of the spin lock was positive, the macro terminates: the kernel
   control path has acquired the spin lock.

---

[*] Spin locks, ironically enough, are global and therefore must themselves be protected against concurrent
accesses.

4. Otherwise, the kernel control path failed in acquiring the spin lock, thus the macro must cycle until the spin lock is released by a kernel control path running on some other CPU. Invokes `preempt_enable()` to undo the increase of the preemption counter done in step 1. If kernel preemption was enabled before executing the `spin_lock` macro, another process can now replace this process while it waits for the spin lock.

5. If the `break_lock` field is equal to zero, sets it to one. By checking this field, the process owning the lock and running on another CPU can learn whether there are other processes waiting for the lock. If a process holds a spin lock for a long time, it may decide to release it prematurely to allow another process waiting for the same spin lock to progress.

6. Executes the wait cycle:

```
while (spin_is_locked(slp) && slp->break_lock)
    cpu_relax();
```

The `cpu_relax()` macro reduces to a `pause` assembly language instruction. This instruction has been introduced in the Pentium 4 model to optimize the execution of spin lock loops. By introducing a short delay, it speeds up the execution of code following the lock and reduces power consumption. The `pause` instruction is backward compatible with earlier models of 80×86 microprocessors because it corresponds to the instruction `rep;nop`, that is, to a no-operation.

7. Jumps back to step 1 to try once more to get the spin lock.

The spin_lock macro without kernel preemption

If the kernel preemption option has not been selected when the kernel was compiled, the `spin_lock` macro is quite different from the one described above. In this case, the macro yields a assembly language fragment that is essentially equivalent to the following tight busy wait:[*]

```
1: lock; decb slp->slock
   jns  3f
2: pause
   cmpb $0,slp->slock
   jle 2b
   jmp 1b
3:
```

The `decb` assembly language instruction decreases the spin lock value; the instruction is atomic because it is prefixed by the `lock` byte. A test is then performed on the sign flag. If it is clear, it means that the spin lock was set to 1 (unlocked), so normal

---

[*] The actual implementation of the tight busy wait loop is slightly more complicated. The code at label 2, which is executed only if the spin lock is busy, is included in an auxiliary section so that in the most frequent case (when the spin lock is already free) the hardware cache is not filled with code that won't be executed. In our discussion, we omit these optimization details.

execution continues at label 3 (the f suffix denotes the fact that the label is a "for-ward" one; it appears in a later line of the program). Otherwise, the tight loop at label 2 (the b suffix denotes a "backward" label) is executed until the spin lock assumes a positive value. Then execution restarts from label 1, since it is unsafe to proceed without checking whether another processor has grabbed the lock.

The spin_unlock macro

The `spin_unlock` macro releases a previously acquired spin lock; it essentially executes the assembly language instruction:

```
movb $1, slp->slock
```

and then invokes `preempt_enable()` (if kernel preemption is not supported, `preempt_enable()` does nothing). Notice that the `lock` byte is not used because write-only accesses in memory are always atomically executed by the current 80×86 microprocessors.

Read/Write Spin Locks

Read/write spin locks have been introduced to increase the amount of concurrency inside the kernel. They allow several kernel control paths to simultaneously read the same data structure, as long as no kernel control path modifies it. If a kernel control path wishes to write to the structure, it must acquire the write version of the read/write lock, which grants exclusive access to the resource. Of course, allowing concurrent reads on data structures improves system performance.

Figure 5-2 illustrates two critical regions (C1 and C2) protected by read/write locks. Kernel control paths R0 and R1 are reading the data structures in C1 at the same time, while W0 is waiting to acquire the lock for writing. Kernel control path W1 is writing the data structures in C2, while both R2 and W2 are waiting to acquire the lock for reading and writing, respectively.
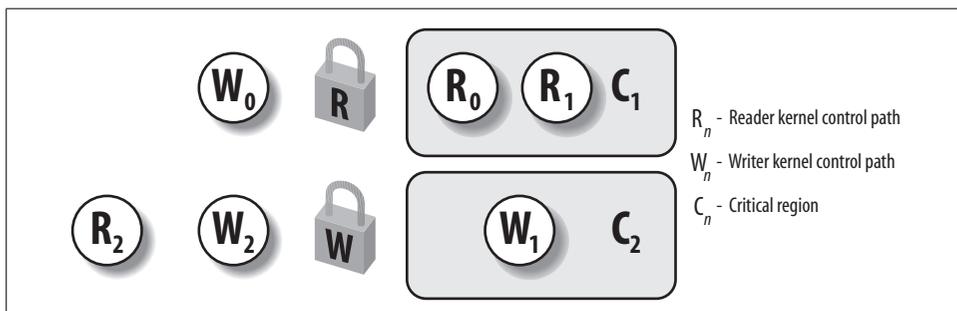
Figure 5-2. Read/write spin locks

Each read/write spin lock is a `rwlock_t` structure; its `lock` field is a 32-bit field that encodes two distinct pieces of information:

- A 24-bit counter denoting the number of kernel control paths currently reading the protected data structure. The two's complement value of this counter is stored in bits 0–23 of the field.

- An unlock flag that is set when no kernel control path is reading or writing, and clear otherwise. This unlock flag is stored in bit 24 of the field.

Notice that the `lock` field stores the number `0x01000000` if the spin lock is idle (unlock flag set and no readers), the number `0x00000000` if it has been acquired for writing (unlock flag clear and no readers), and any number in the sequence `0x00ffffff`, `0x00fffffe`, and so on, if it has been acquired for reading by one, two, or more processes (unlock flag clear and the two's complement on 24 bits of the number of readers). As the `spinlock_t` structure, the `rwlock_t` structure also includes a `break_lock` field.

The `rwlock_init` macro initializes the `lock` field of a read/write spin lock to `0x01000000` (unlocked) and the `break_lock` field to zero.

Getting and releasing a lock for reading

The `read_lock` macro, applied to the address `rwlp` of a read/write spin lock, is similar to the `spin_lock` macro described in the previous section. If the kernel preemption option has been selected when the kernel was compiled, the macro performs the very same actions as those of `spin_lock()`, with just one exception: to effectively acquire the read/write spin lock in step 2, the macro executes the `_raw_read_trylock()` function:

```
int _raw_read_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    atomic_dec(count);
    if (atomic_read(count) >= 0)
        return 1;
    atomic_inc(count);
    return 0;
}
```

The `lock` field—the read/write lock counter—is accessed by means of atomic operations. Notice, however, that the whole function does not act atomically on the counter: for instance, the counter might change after having tested its value with the `if` statement and before returning 1. Nevertheless, the function works properly: in fact, the function returns 1 only if the counter was not zero or negative before the decrement, because the counter is equal to `0x01000000` for no owner, `0x00ffffff` for one reader, and `0x00000000` for one writer.

If the kernel preemption option has not been selected when the kernel was compiled, the `read_lock` macro yields the following assembly language code:

```
    movl $rwlp->lock,%eax
    lock; subl $1,(%eax)
    jns 1f
    call __read_lock_failed
1:
```

where `__read_lock_failed()` is the following assembly language function:

```
__read_lock_failed:
    lock; incl (%eax)
1:  pause
    cmpl $1,(%eax)
    js 1b
    lock; decl (%eax)
    js __read_lock_failed
    ret
```

The `read_lock` macro atomically decreases the spin lock value by 1, thus increasing the number of readers. The spin lock is acquired if the decrement operation yields a nonnegative value; otherwise, the `__read_lock_failed()` function is invoked. The function atomically increases the `lock` field to undo the decrement operation performed by the `read_lock` macro, and then loops until the field becomes positive (greater than or equal to 1). Next, `__read_lock_failed()` tries to get the spin lock again (another kernel control path could acquire the spin lock for writing right after the `cmpl` instruction).

Releasing the read lock is quite simple, because the `read_unlock` macro must simply increase the counter in the `lock` field with the assembly language instruction:

```
    lock; incl rwlp->lock
```

to decrease the number of readers, and then invoke `preempt_enable()` to reenable kernel preemption.

Getting and releasing a lock for writing

The `write_lock` macro is implemented in the same way as `spin_lock()` and `read_lock()`. For instance, if kernel preemption is supported, the function disables kernel preemption and tries to grab the lock right away by invoking `_raw_write_trylock()`. If this function returns 0, the lock was already taken, thus the macro reenables kernel preemption and starts a busy wait loop, as explained in the description of `spin_lock()` in the previous section.

The `_raw_write_trylock()` function is shown below:

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

The `_raw_write_trylock()` function subtracts `0x01000000` from the read/write spin lock value, thus clearing the unlock flag (bit 24). If the subtraction operation yields zero (no readers), the lock is acquired and the function returns 1; otherwise, the function atomically adds `0x01000000` to the spin lock value to undo the subtraction operation.

Once again, releasing the write lock is much simpler because the `write_unlock` macro must simply set the unlock flag in the `lock` field with the assembly language instruction:

```
lock; addl $0x01000000,rwlp
```

and then invoke `preempt_enable()`.

Seqlocks

When using read/write spin locks, requests issued by kernel control paths to perform a `read_lock` or a `write_lock` operation have the same priority: readers must wait until the writer has finished and, similarly, a writer must wait until all readers have finished.

Seqlocks introduced in Linux 2.6 are similar to read/write spin locks, except that they give a much higher priority to writers: in fact a writer is allowed to proceed even when readers are active. The good part of this strategy is that a writer never waits (unless another writer is active); the bad part is that a reader may sometimes be forced to read the same data several times until it gets a valid copy.

Each seqlock is a `seqlock_t` structure consisting of two fields: a `lock` field of type `spinlock_t` and an integer `sequence` field. This second field plays the role of a sequence counter. Each reader must read this sequence counter twice, before and after reading the data, and check whether the two values coincide. In the opposite case, a new writer has become active and has increased the sequence counter, thus implicitly telling the reader that the data just read is not valid.

A `seqlock_t` variable is initialized to "unlocked" either by assigning to it the value `SEQLOCK_UNLOCKED`, or by executing the `seqlock_init` macro. Writers acquire and release a seqlock by invoking `write_seqlock()` and `write_sequnlock()`. The first function acquires the spin lock in the `seqlock_t` data structure, then increases the sequence counter by one; the second function increases the sequence counter once more, then releases the spin lock. This ensures that when the writer is in the middle of writing, the counter is odd, and that when no writer is altering data, the counter is even. Readers implement a critical region as follows:

```
unsigned int seq;
do {
    seq = read_seqbegin(&seqlock);
    /* ... CRITICAL REGION ... */
} while (read_seqretry(&seqlock, seq));
```

read_seqbegin() returns the current sequence number of the seqlock; read_seqretry() returns 1 if either the value of the seq local variable is odd (a writer was updating the data structure when the read_seqbegin() function has been invoked), or if the value of seq does not match the current value of the seqlock's sequence counter (a writer started working while the reader was still executing the code in the critical region).

Notice that when a reader enters a critical region, it does not need to disable kernel preemption; on the other hand, the writer automatically disables kernel preemption when entering the critical region, because it acquires the spin lock.

Not every kind of data structure can be protected by a seqlock. As a general rule, the following conditions must hold:

- The data structure to be protected does not include pointers that are modified by the writers and dereferenced by the readers (otherwise, a writer could change the pointer under the nose of the readers)
- The code in the critical regions of the readers does not have side effects (otherwise, multiple reads would have different effects from a single read)

Furthermore, the critical regions of the readers should be short and writers should seldom acquire the seqlock, otherwise repeated read accesses would cause a severe overhead. A typical usage of seqlocks in Linux 2.6 consists of protecting some data structures related to the system time handling (see Chapter 6).

Read-Copy Update (RCU)

Read-copy update (*RCU*) is yet another synchronization technique designed to protect data structures that are mostly accessed for reading by several CPUs. RCU allows many readers and many writers to proceed concurrently (an improvement over seqlocks, which allow only one writer to proceed). Moreover, RCU is lock-free, that is, it uses no lock or counter shared by all CPUs; this is a great advantage over read/write spin locks and seqlocks, which have a high overhead due to cache line-snooping and invalidation.

How does RCU obtain the surprising result of synchronizing several CPUs without shared data structures? The key idea consists of limiting the scope of RCU as follows:

1. Only data structures that are dynamically allocated and referenced by means of pointers can be protected by RCU.
2. No kernel control path can sleep inside a critical region protected by RCU.

When a kernel control path wants to read an RCU-protected data structure, it executes the rcu_read_lock() macro, which is equivalent to preempt_disable(). Next, the reader dereferences the pointer to the data structure and starts reading it. As stated above, the reader cannot sleep until it finishes reading the data structure; the end of the critical region is marked by the rcu_read_unlock() macro, which is equivalent to preempt_enable().

Because the reader does very little to prevent race conditions, we could expect that the writer has to work a bit more. In fact, when a writer wants to update the data structure, it dereferences the pointer and makes a copy of the whole data structure. Next, the writer modifies the copy. Once finished, the writer changes the pointer to the data structure so as to make it point to the updated copy. Because changing the value of the pointer is an atomic operation, each reader or writer sees either the old copy or the new one: no corruption in the data structure may occur. However, a memory barrier is required to ensure that the updated pointer is seen by the other CPUs only after the data structure has been modified. Such a memory barrier is implicitly introduced if a spin lock is coupled with RCU to forbid the concurrent execution of writers.

The real problem with the RCU technique, however, is that the old copy of the data structure cannot be freed right away when the writer updates the pointer. In fact, the readers that were accessing the data structure when the writer started its update could still be reading the old copy. The old copy can be freed only after all (potential) readers on the CPUs have executed the `rcu_read_unlock()` macro. The kernel requires every potential reader to execute that macro before:

- The CPU performs a process switch (see restriction 2 earlier).
- The CPU starts executing in User Mode.
- The CPU executes the idle loop (see the section "Kernel Threads" in Chapter 3).

In each of these cases, we say that the CPU has gone through a *quiescent state*.

The `call_rcu()` function is invoked by the writer to get rid of the old copy of the data structure. It receives as its parameters the address of an `rcu_head` descriptor (usually embedded inside the data structure to be freed) and the address of a *callback* function to be invoked when all CPUs have gone through a quiescent state. Once executed, the callback function usually frees the old copy of the data structure.

The `call_rcu()` function stores in the `rcu_head` descriptor the address of the callback and its parameter, then inserts the descriptor in a per-CPU list of callbacks. Periodically, once every tick (see the section "Updating Local CPU Statistics" in Chapter 6), the kernel checks whether the local CPU has gone through a quiescent state. When all CPUs have gone through a quiescent state, a local tasklet—whose descriptor is stored in the `rcu_tasklet` per-CPU variable—executes all callbacks in the list.

RCU is a new addition in Linux 2.6; it is used in the networking layer and in the Virtual Filesystem.

Semaphores

We have already introduced semaphores in the section "Synchronization and Critical Regions" in Chapter 1. Essentially, they implement a locking primitive that allows waiters to sleep until the desired resource becomes free.

Actually, Linux offers two kinds of semaphores:

- Kernel semaphores, which are used by kernel control paths
- System V IPC semaphores, which are used by User Mode processes

In this section, we focus on kernel semaphores, while IPC semaphores are described in Chapter 19.

A kernel semaphore is similar to a spin lock, in that it doesn't allow a kernel control path to proceed unless the lock is open. However, whenever a kernel control path tries to acquire a busy resource protected by a kernel semaphore, the corresponding process is suspended. It becomes runnable again when the resource is released. Therefore, kernel semaphores can be acquired only by functions that are allowed to sleep; interrupt handlers and deferrable functions cannot use them.

A kernel semaphore is an object of type `struct semaphore`, containing the fields shown in the following list.

count

> Stores an `atomic_t` value. If it is greater than 0, the resource is free—that is, it is currently available. If count is equal to 0, the semaphore is busy but no other process is waiting for the protected resource. Finally, if count is negative, the resource is unavailable and at least one process is waiting for it.

wait

> Stores the address of a wait queue list that includes all sleeping processes that are currently waiting for the resource. Of course, if count is greater than or equal to 0, the wait queue is empty.

sleepers

> Stores a flag that indicates whether some processes are sleeping on the semaphore. We'll see this field in operation soon.

The `init_MUTEX()` and `init_MUTEX_LOCKED()` functions may be used to initialize a semaphore for exclusive access: they set the count field to 1 (free resource with exclusive access) and 0 (busy resource with exclusive access currently granted to the process that initializes the semaphore), respectively. The `DECLARE_MUTEX` and `DECLARE_MUTEX_LOCKED` macros do the same, but they also statically allocate the `struct semaphore` variable. Note that a semaphore could also be initialized with an arbitrary positive value n for count. In this case, at most n processes are allowed to concurrently access the resource.

Getting and releasing semaphores

Let's start by discussing how to release a semaphore, which is much simpler than getting one. When a process wishes to release a kernel semaphore lock, it invokes the

up() function. This function is essentially equivalent to the following assembly language fragment:

```
        movl $sem->count,%ecx
        lock; incl (%ecx)
        jg 1f
        lea %ecx,%eax
        pushl %edx
        pushl %ecx
        call __up
        popl %ecx
        popl %edx
    1:
```

where __up() is the following C function:

```
    __attribute__((regparm(3))) void __up(struct semaphore *sem)
    {
        wake_up(&sem->wait);
    }
```

The up() function increases the count field of the *sem semaphore, and then it checks whether its value is greater than 0. The increment of count and the setting of the flag tested by the following jump instruction must be atomically executed, or else another kernel control path could concurrently access the field value, with disastrous results. If count is greater than 0, there was no process sleeping in the wait queue, so nothing has to be done. Otherwise, the __up() function is invoked so that one sleeping process is woken up. Notice that __up() receives its parameter from the eax register (see the description of the __switch_to() function in the section "Performing the Process Switch" in Chapter 3).

Conversely, when a process wishes to acquire a kernel semaphore lock, it invokes the down() function. The implementation of down() is quite involved, but it is essentially equivalent to the following:

```
    down:
        movl $sem->count,%ecx
        lock; decl (%ecx);
        jns 1f
        lea %ecx, %eax
        pushl %edx
        pushl %ecx
        call __down
        popl %ecx
        popl %edx
    1:
```

where __down() is the following C function:

```
    __attribute__((regparm(3))) void __down(struct semaphore * sem)
    {
        DECLARE_WAITQUEUE(wait, current);
        unsigned long flags;
        current->state = TASK_UNINTERRUPTIBLE;
```

```
        spin_lock_irqsave(&sem->wait.lock, flags);
        add_wait_queue_exclusive_locked(&sem->wait, &wait);
        sem->sleepers++;
        for (;;) {
            if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
                sem->sleepers = 0;
                break;
            }
            sem->sleepers = 1;
            spin_unlock_irqrestore(&sem->wait.lock, flags);
            schedule();
            spin_lock_irqsave(&sem->wait.lock, flags);
            current->state = TASK_UNINTERRUPTIBLE;
        }
        remove_wait_queue_locked(&sem->wait, &wait);
        wake_up_locked(&sem->wait);
        spin_unlock_irqrestore(&sem->wait.lock, flags);
        current->state = TASK_RUNNING;
    }
```

The `down()` function decreases the `count` field of the *sem semaphore, and then checks whether its value is negative. Again, the decrement and the test must be atomically executed. If `count` is greater than or equal to 0, the current process acquires the resource and the execution continues normally. Otherwise, `count` is negative, and the current process must be suspended. The contents of some registers are saved on the stack, and then `__down()` is invoked.

Essentially, the `__down()` function changes the state of the current process from `TASK_RUNNING` to `TASK_UNINTERRUPTIBLE`, and it puts the process in the semaphore wait queue. Before accessing the fields of the `semaphore` structure, the function also gets the `sem->wait.lock` spin lock that protects the semaphore wait queue (see "How Processes Are Organized" in Chapter 3) and disables local interrupts. Usually, wait queue functions get and release the wait queue spin lock as necessary when inserting and deleting an element. The `__down()` function, however, uses the wait queue spin lock also to protect the other fields of the `semaphore` data structure, so that no process running on another CPU is able to read or modify them. To that end, `__down()` uses the "_locked" versions of the wait queue functions, which assume that the spin lock has been already acquired before their invocations.

The main task of the `__down()` function is to suspend the current process until the semaphore is released. However, the way in which this is done is quite involved. To easily understand the code, keep in mind that the `sleepers` field of the semaphore is usually set to 0 if no process is sleeping in the wait queue of the semaphore, and it is set to 1 otherwise. Let's try to explain the code by considering a few typical cases.

MUTEX semaphore open (`count` equal to 1, `sleepers` equal to 0)
    The `down` macro just sets the `count` field to 0 and jumps to the next instruction of the main program; therefore, the `__down()` function is not executed at all.

*MUTEX semaphore closed, no sleeping processes (*count *equal to 0,* sleepers *equal to 0)*
> The down macro decreases count and invokes the __down() function with the count field set to −1 and the sleepers field set to 0. In each iteration of the loop, the function checks whether the count field is negative. (Observe that the count field is not changed by atomic_add_negative() because sleepers is equal to 0 when the function is invoked.)
>
> - If the count field is negative, the function invokes schedule() to suspend the current process. The count field is still set to −1, and the sleepers field to 1. The process picks up its run subsequently inside this loop and issues the test again.
> - If the count field is not negative, the function sets sleepers to 0 and exits from the loop. It tries to wake up another process in the semaphore wait queue (but in our scenario, the queue is now empty) and terminates holding the semaphore. On exit, both the count field and the sleepers field are set to 0, as required when the semaphore is closed but no process is waiting for it.

*MUTEX semaphore closed, other sleeping processes (*count *equal to −1,* sleepers *equal to 1)*
> The down macro decreases count and invokes the __down() function with count set to −2 and sleepers set to 1. The function temporarily sets sleepers to 2, and then undoes the decrement performed by the down macro by adding the value sleepers−1 to count. At the same time, the function checks whether count is still negative (the semaphore could have been released by the holding process right before __down() entered the critical region).
>
> - If the count field is negative, the function resets sleepers to 1 and invokes schedule() to suspend the current process. The count field is still set to −1, and the sleepers field to 1.
> - If the count field is not negative, the function sets sleepers to 0, tries to wake up another process in the semaphore wait queue, and exits holding the semaphore. On exit, the count field is set to 0 and the sleepers field to 0. The values of both fields look wrong, because there are other sleeping processes. However, consider that another process in the wait queue has been woken up. This process does another iteration of the loop; the atomic_add_negative() function subtracts 1 from count, restoring it to −1; moreover, before returning to sleep, the woken-up process resets sleepers to 1.

So, the code properly works in all cases. Consider that the wake_up() function in __down() wakes up at most one process, because the sleeping processes in the wait queue are exclusive (see the section "How Processes Are Organized" in Chapter 3).

Only exception handlers, and particularly system call service routines, can use the down() function. Interrupt handlers or deferrable functions must not invoke down(), because this function suspends the process when the semaphore is busy. For this reason, Linux provides the down_trylock() function, which may be safely used by one

of the previously mentioned asynchronous functions. It is identical to `down()` except when the resource is busy. In this case, the function returns immediately instead of putting the process to sleep.

A slightly different function called `down_interruptible()` is also defined. It is widely used by device drivers, because it allows processes that receive a signal while being blocked on a semaphore to give up the "down" operation. If the sleeping process is woken up by a signal before getting the needed resource, the function increases the `count` field of the semaphore and returns the value -EINTR. On the other hand, if `down_interruptible()` runs to normal completion and gets the resource, it returns 0. The device driver may thus abort the I/O operation when the return value is -EINTR.

Finally, because processes usually find semaphores in an open state, the semaphore functions are optimized for this case. In particular, the `up()` function does not execute jump instructions if the semaphore wait queue is empty; similarly, the `down()` function does not execute jump instructions if the semaphore is open. Much of the complexity of the semaphore implementation is precisely due to the effort of avoiding costly instructions in the main branch of the execution flow.

Read/Write Semaphores

Read/write semaphores are similar to the read/write spin locks described earlier in the section "Read/Write Spin Locks," except that waiting processes are suspended instead of spinning until the semaphore becomes open again.

Many kernel control paths may concurrently acquire a read/write semaphore for reading; however, every writer kernel control path must have exclusive access to the protected resource. Therefore, the semaphore can be acquired for writing only if no other kernel control path is holding it for either read or write access. Read/write semaphores improve the amount of concurrency inside the kernel and improve overall system performance.

The kernel handles all processes waiting for a read/write semaphore in strict FIFO order. Each reader or writer that finds the semaphore closed is inserted in the last position of a semaphore's wait queue list. When the semaphore is released, the process in the first position of the wait queue list are checked. The first process is always awoken. If it is a writer, the other processes in the wait queue continue to sleep. If it is a reader, all readers at the start of the queue, up to the first writer, are also woken up and get the lock. However, readers that have been queued after a writer continue to sleep.

Each read/write semaphore is described by a `rw_semaphore` structure that includes the following fields:

count
    Stores two 16-bit counters. The counter in the most significant word encodes in two's complement form the sum of the number of nonwaiting writers (either 0

or 1) and the number of waiting kernel control paths. The counter in the less significant word encodes the total number of nonwaiting readers and writers.

wait_list

Points to a list of waiting processes. Each element in this list is a `rwsem_waiter` structure, including a pointer to the descriptor of the sleeping process and a flag indicating whether the process wants the semaphore for reading or for writing.

wait_lock

A spin lock used to protect the wait queue list and the `rw_semaphore` structure itself.

The `init_rwsem()` function initializes an `rw_semaphore` structure by setting the `count` field to 0, the `wait_lock` spin lock to unlocked, and `wait_list` to the empty list.

The `down_read()` and `down_write()` functions acquire the read/write semaphore for reading and writing, respectively. Similarly, the `up_read()` and `up_write()` functions release a read/write semaphore previously acquired for reading and for writing. The `down_read_trylock()` and `down_write_trylock()` functions are similar to `down_read()` and `down_write()`, respectively, but they do not block the process if the semaphore is busy. Finally, the `downgrade_write()` function atomically transforms a write lock into a read lock. The implementation of these five functions is long, but easy to follow because it resembles the implementation of normal semaphores; therefore, we avoid describing them.

Completions

Linux 2.6 also makes use of another synchronization primitive similar to semaphores: *completions*. They have been introduced to solve a subtle race condition that occurs in multiprocessor systems when process A allocates a temporary semaphore variable, initializes it as closed MUTEX, passes its address to process B, and then invokes `down()` on it. Process A plans to destroy the semaphore as soon as it awakens. Later on, process B running on a different CPU invokes `up()` on the semaphore. However, in the current implementation `up()` and `down()` can execute concurrently on the same semaphore. Thus, process A can be woken up and destroy the temporary semaphore while process B is still executing the `up()` function. As a result, `up()` might attempt to access a data structure that no longer exists.

Of course, it is possible to change the implementation of `down()` and `up()` to forbid concurrent executions on the same semaphore. However, this change would require additional instructions, which is a bad thing to do for functions that are so heavily used.

The completion is a synchronization primitive that is specifically designed to solve this problem. The `completion` data structure includes a wait queue head and a flag:

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

The function corresponding to `up()` is called `complete()`. It receives as an argument the address of a `completion` data structure, invokes `spin_lock_irqsave()` on the spin lock of the completion's wait queue, increases the `done` field, wakes up the exclusive process sleeping in the `wait` wait queue, and finally invokes `spin_unlock_irqrestore()`.

The function corresponding to `down()` is called `wait_for_completion()`. It receives as an argument the address of a `completion` data structure and checks the value of the `done` flag. If it is greater than zero, `wait_for_completion()` terminates, because `complete()` has already been executed on another CPU. Otherwise, the function adds `current` to the tail of the wait queue as an exclusive process and puts `current` to sleep in the `TASK_UNINTERRUPTIBLE` state. Once woken up, the function removes `current` from the wait queue. Then, the function checks the value of the `done` flag: if it is equal to zero the function terminates, otherwise, the current process is suspended again. As in the case of the `complete()` function, `wait_for_completion()` makes use of the spin lock in the completion's wait queue.

The real difference between completions and semaphores is how the spin lock included in the wait queue is used. In completions, the spin lock is used to ensure that `complete()` and `wait_for_completion()` cannot execute concurrently. In semaphores, the spin lock is used to avoid letting concurrent `down()`'s functions mess up the `semaphore` data structure.

Local Interrupt Disabling

Interrupt disabling is one of the key mechanisms used to ensure that a sequence of kernel statements is treated as a critical section. It allows a kernel control path to continue executing even when hardware devices issue IRQ signals, thus providing an effective way to protect data structures that are also accessed by interrupt handlers. By itself, however, local interrupt disabling does not protect against concurrent accesses to data structures by interrupt handlers running on other CPUs, so in multiprocessor systems, local interrupt disabling is often coupled with spin locks (see the later section "Synchronizing Accesses to Kernel Data Structures").

The `local_irq_disable()` macro, which makes use of the `cli` assembly language instruction, disables interrupts on the local CPU. The `local_irq_enable()` macro, which makes use of the of the `sti` assembly language instruction, enables them. As stated in the section "IRQs and Interrupts" in Chapter 4, the `cli` and `sti` assembly language instructions, respectively, clear and set the `IF` flag of the `eflags` control register. The `irqs_disabled()` macro yields the value one if the `IF` flag of the `eflags` register is clear, the value one if the flag is set.

When the kernel enters a critical section, it disables interrupts by clearing the `IF` flag of the `eflags` register. But at the end of the critical section, often the kernel can't simply set the flag again. Interrupts can execute in nested fashion, so the kernel does not necessarily know what the `IF` flag was before the current control path executed. In these cases, the control path must save the old setting of the flag and restore that setting at the end.

Saving and restoring the `eflags` content is achieved by means of the `local_irq_save` and `local_irq_restore` macros, respectively. The `local_irq_save` macro copies the content of the `eflags` register into a local variable; the `IF` flag is then cleared by a `cli` assembly language instruction. At the end of the critical region, the macro `local_irq_restore` restores the original content of `eflags`; therefore, interrupts are enabled only if they were enabled before this control path issued the `cli` assembly language instruction.

Disabling and Enabling Deferrable Functions

In the section "Softirqs" in Chapter 4, we explained that deferrable functions can be executed at unpredictable times (essentially, on termination of hardware interrupt handlers). Therefore, data structures accessed by deferrable functions must be protected against race conditions.

A trivial way to forbid deferrable functions execution on a CPU is to disable interrupts on that CPU. Because no interrupt handler can be activated, softirq actions cannot be started asynchronously.

As we'll see in the next section, however, the kernel sometimes needs to disable deferrable functions without disabling interrupts. Local deferrable functions can be enabled or disabled on the local CPU by acting on the softirq counter stored in the `preempt_count` field of the current's `thread_info` descriptor.

Recall that the `do_softirq()` function never executes the softirqs if the softirq counter is positive. Moreover, tasklets are implemented on top of softirqs, so setting this counter to a positive value disables the execution of all deferrable functions on a given CPU, not just softirqs.

The `local_bh_disable` macro adds one to the softirq counter of the local CPU, while the `local_bh_enable()` function subtracts one from it. The kernel can thus use several nested invocations of `local_bh_disable`; deferrable functions will be enabled again only by the `local_bh_enable` macro matching the first `local_bh_disable` invocation.

After having decreased the softirq counter, `local_bh_enable()` performs two important operations that help to ensure timely execution of long-waiting threads:

1. Checks the hardirq counter and the softirq counter in the `preempt_count` field of the local CPU; if both of them are zero and there are pending softirqs to be executed, invokes `do_softirq()` to activate them (see the section "Softirqs" in Chapter 4).

2. Checks whether the `TIF_NEED_RESCHED` flag of the local CPU is set; if so, a process switch request is pending, thus invokes the `preempt_schedule()` function (see the section "Kernel Preemption" earlier in this chapter).

Synchronizing Accesses to Kernel Data Structures

A shared data structure can be protected against race conditions by using some of the synchronization primitives shown in the previous section. Of course, system performance may vary considerably, depending on the kind of synchronization primitive selected. Usually, the following rule of thumb is adopted by kernel developers: *always keep the concurrency level as high as possible in the system.*

In turn, the concurrency level in the system depends on two main factors:

- The number of I/O devices that operate concurrently
- The number of CPUs that do productive work

To maximize I/O throughput, interrupts should be disabled for very short periods of time. As described in the section "IRQs and Interrupts" in Chapter 4, when interrupts are disabled, IRQs issued by I/O devices are temporarily ignored by the PIC, and no new activity can start on such devices.

To use CPUs efficiently, synchronization primitives based on spin locks should be avoided whenever possible. When a CPU is executing a tight instruction loop waiting for the spin lock to open, it is wasting precious machine cycles. Even worse, as we have already said, spin locks have negative effects on the overall performance of the system because of their impact on the hardware caches.

Let's illustrate a couple of cases in which synchronization can be achieved while still maintaining a high concurrency level:

- A shared data structure consisting of a single integer value can be updated by declaring it as an `atomic_t` type and by using atomic operations. An atomic operation is faster than spin locks and interrupt disabling, and it slows down only kernel control paths that concurrently access the data structure.

- Inserting an element into a shared linked list is never atomic, because it consists of at least two pointer assignments. Nevertheless, the kernel can sometimes perform this insertion operation without using locks or disabling interrupts. As an example of why this works, we'll consider the case where a system call service routine (see "System Call Handler and Service Routines" in Chapter 10) inserts new elements in a singly linked list, while an interrupt handler or deferrable function asynchronously looks up the list.

  In the C language, insertion is implemented by means of the following pointer assignments:

  ```
  new->next = list_element->next;
  list_element->next = new;
  ```

  In assembly language, insertion reduces to two consecutive atomic instructions. The first instruction sets up the `next` pointer of the `new` element, but it does not modify the list. Thus, if the interrupt handler sees the list between the execution of the first and second instructions, it sees the list without the new element. If

the handler sees the list after the execution of the second instruction, it sees the list with the new element. The important point is that in either case, the list is consistent and in an uncorrupted state. However, this integrity is assured only if the interrupt handler does not modify the list. If it does, the next pointer that was just set within the new element might become invalid.

However, developers must ensure that the order of the two assignment operations cannot be subverted by the compiler or the CPU's control unit; otherwise, if the system call service routine is interrupted by the interrupt handler between the two assignments, the handler finds a corrupted list. Therefore, a write memory barrier primitive is required:

```
new->next = list_element->next;
wmb();
list_element->next = new;
```

Choosing Among Spin Locks, Semaphores, and Interrupt Disabling

Unfortunately, access patterns to most kernel data structures are a lot more complex than the simple examples just shown, and kernel developers are forced to use semaphores, spin locks, interrupts, and softirq disabling. Generally speaking, choosing the synchronization primitives depends on what kinds of kernel control paths access the data structure, as shown in Table 5-8. Remember that whenever a kernel control path acquires a spin lock (as well as a read/write lock, a seqlock, or a RCU "read lock"), disables the local interrupts, or disables the local softirqs, kernel preemption is automatically disabled.

Table 5-8. Protection required by data structures accessed by kernel control paths

Kernel control paths accessing the data structure	UP protection	MP further protection
Exceptions	Semaphore	None
Interrupts	Local interrupt disabling	Spin lock
Deferrable functions	None	None or spin lock (see Table 5-9)
Exceptions + Interrupts	Local interrupt disabling	Spin lock
Exceptions + Deferrable functions	Local softirq disabling	Spin lock
Interrupts + Deferrable functions	Local interrupt disabling	Spin lock
Exceptions + Interrupts + Deferrable functions	Local interrupt disabling	Spin lock

Protecting a data structure accessed by exceptions

When a data structure is accessed only by exception handlers, race conditions are usually easy to understand and prevent. The most common exceptions that give rise to synchronization problems are the system call service routines (see the section "System Call Handler and Service Routines" in Chapter 10) in which the CPU operates in Kernel Mode to offer a service to a User Mode program. Thus, a data structure

accessed only by an exception usually represents a resource that can be assigned to one or more processes.

Race conditions are avoided through semaphores, because these primitives allow the process to sleep until the resource becomes available. Notice that semaphores work equally well both in uniprocessor and multiprocessor systems.

Kernel preemption does not create problems either. If a process that owns a semaphore is preempted, a new process running on the same CPU could try to get the semaphore. When this occurs, the new process is put to sleep, and eventually the old process will release the semaphore. The only case in which kernel preemption must be explicitly disabled is when accessing per-CPU variables, as explained in the section "Per-CPU Variables" earlier in this chapter.

Protecting a data structure accessed by interrupts

Suppose that a data structure is accessed by only the "top half" of an interrupt handler. We learned in the section "Interrupt Handling" in Chapter 4 that each interrupt handler is serialized with respect to itself—that is, it cannot execute more than once concurrently. Thus, accessing the data structure does not require synchronization primitives.

Things are different, however, if the data structure is accessed by several interrupt handlers. A handler may interrupt another handler, and different interrupt handlers may run concurrently in multiprocessor systems. Without synchronization, the shared data structure might easily become corrupted.

In uniprocessor systems, race conditions must be avoided by disabling interrupts in all critical regions of the interrupt handler. Nothing less will do because no other synchronization primitives accomplish the job. A semaphore can block the process, so it cannot be used in an interrupt handler. A spin lock, on the other hand, can freeze the system: if the handler accessing the data structure is interrupted, it cannot release the lock; therefore, the new interrupt handler keeps waiting on the tight loop of the spin lock.

Multiprocessor systems, as usual, are even more demanding. Race conditions cannot be avoided by simply disabling local interrupts. In fact, even if interrupts are disabled on a CPU, interrupt handlers can still be executed on the other CPUs. The most convenient method to prevent the race conditions is to disable local interrupts (so that other interrupt handlers running on the same CPU won't interfere) *and* to acquire a spin lock or a read/write spin lock that protects the data structure. Notice that these additional spin locks cannot freeze the system because even if an interrupt handler finds the lock closed, eventually the interrupt handler on the other CPU that owns the lock will release it.

The Linux kernel uses several macros that couple the enabling and disabling of local interrupts with spin lock handling. Table 5-9 describes all of them. In uniprocessor systems, these macros just enable or disable local interrupts and kernel preemption.

Table 5-9. Interrupt-aware spin lock macros

Macro	Description
spin_lock_irq(l)	local_irq_disable(); spin_lock(l)
spin_unlock_irq(l)	spin_unlock(l); local_irq_enable()
spin_lock_bh(l)	local_bh_disable(); spin_lock(l)
spin_unlock_bh(l)	spin_unlock(l); local_bh_enable()
spin_lock_irqsave(l,f)	local_irq_save(f); spin_lock(l)
spin_unlock_irqrestore(l,f)	spin_unlock(l); local_irq_restore(f)
read_lock_irq(l)	local_irq_disable(); read_lock(l)
read_unlock_irq(l)	read_unlock(l); local_irq_enable()
read_lock_bh(l)	local_bh_disable(); read_lock(l)
read_unlock_bh(l)	read_unlock(l); local_bh_enable()
write_lock_irq(l)	local_irq_disable(); write_lock(l)
write_unlock_irq(l)	write_unlock(l); local_irq_enable()
write_lock_bh(l)	local_bh_disable(); write_lock(l)
write_unlock_bh(l)	write_unlock(l); local_bh_enable()
read_lock_irqsave(l,f)	local_irq_save(f); read_lock(l)
read_unlock_irqrestore(l,f)	read_unlock(l); local_irq_restore(f)
write_lock_irqsave(l,f)	local_irq_save(f); write_lock(l)
write_unlock_irqrestore(l,f)	write_unlock(l); local_irq_restore(f)
read_seqbegin_irqsave(l,f)	local_irq_save(f); read_seqbegin(l)
read_seqretry_irqrestore(l,v,f)	read_seqretry(l,v); local_irq_restore(f)
write_seqlock_irqsave(l,f)	local_irq_save(f); write_seqlock(l)
write_sequnlock_irqrestore(l,f)	write_sequnlock(l); local_irq_restore(f)
write_seqlock_irq(l)	local_irq_disable(); write_seqlock(l)
write_sequnlock_irq(l)	write_sequnlock(l); local_irq_enable()
write_seqlock_bh(l)	local_bh_disable(); write_seqlock(l);
write_sequnlock_bh(l)	write_sequnlock(l); local_bh_enable()

Protecting a data structure accessed by deferrable functions

What kind of protection is required for a data structure accessed only by deferrable functions? Well, it mostly depends on the kind of deferrable function. In the section "Softirqs and Tasklets" in Chapter 4, we explained that softirqs and tasklets essentially differ in their degree of concurrency.

First of all, no race condition may exist in uniprocessor systems. This is because execution of deferrable functions is always serialized on a CPU—that is, a deferrable function cannot be interrupted by another deferrable function. Therefore, no synchronization primitive is ever required.

Conversely, in multiprocessor systems, race conditions do exist because several deferrable functions may run concurrently. Table 5-10 lists all possible cases.

Table 5-10. Protection required by data structures accessed by deferrable functions in SMP

Deferrable functions accessing the data structure	Protection
Softirqs	Spin lock
One tasklet	None
Many tasklets	Spin lock

A data structure accessed by a softirq must always be protected, usually by means of a spin lock, because the same softirq may run concurrently on two or more CPUs. Conversely, a data structure accessed by just one kind of tasklet need not be protected, because tasklets of the same kind cannot run concurrently. However, if the data structure is accessed by several kinds of tasklets, then it must be protected.

Protecting a data structure accessed by exceptions and interrupts

Let's consider now a data structure that is accessed both by exceptions (for instance, system call service routines) and interrupt handlers.

On uniprocessor systems, race condition prevention is quite simple, because interrupt handlers are not reentrant and cannot be interrupted by exceptions. As long as the kernel accesses the data structure with local interrupts disabled, the kernel cannot be interrupted when accessing the data structure. However, if the data structure is accessed by just one kind of interrupt handler, the interrupt handler can freely access the data structure without disabling local interrupts.

On multiprocessor systems, we have to take care of concurrent executions of exceptions and interrupts on other CPUs. Local interrupt disabling must be coupled with a spin lock, which forces the concurrent kernel control paths to wait until the handler accessing the data structure finishes its work.

Sometimes it might be preferable to replace the spin lock with a semaphore. Because interrupt handlers cannot be suspended, they must acquire the semaphore using a tight loop and the down_trylock() function; for them, the semaphore acts essentially as a spin lock. System call service routines, on the other hand, may suspend the calling processes when the semaphore is busy. For most system calls, this is the expected behavior. In this case, semaphores are preferable to spin locks, because they lead to a higher degree of concurrency of the system.

Protecting a data structure accessed by exceptions and deferrable functions

A data structure accessed both by exception handlers and deferrable functions can be treated like a data structure accessed by exception and interrupt handlers. In fact, deferrable functions are essentially activated by interrupt occurrences, and no exception can be raised while a deferrable function is running. Coupling local interrupt disabling with a spin lock is therefore sufficient.

Actually, this is much more than sufficient: the exception handler can simply disable deferrable functions instead of local interrupts by using the `local_bh_disable()` macro (see the section "Softirqs" in Chapter 4). Disabling only the deferrable functions is preferable to disabling interrupts, because interrupts continue to be serviced by the CPU. Execution of deferrable functions on each CPU is serialized, so no race condition exists.

As usual, in multiprocessor systems, spin locks are required to ensure that the data structure is accessed at any time by just one kernel control.

Protecting a data structure accessed by interrupts and deferrable functions

This case is similar to that of a data structure accessed by interrupt and exception handlers. An interrupt might be raised while a deferrable function is running, but no deferrable function can stop an interrupt handler. Therefore, race conditions must be avoided by disabling local interrupts during the deferrable function. However, an interrupt handler can freely touch the data structure accessed by the deferrable function without disabling interrupts, provided that no other interrupt handler accesses that data structure.

Again, in multiprocessor systems, a spin lock is always required to forbid concurrent accesses to the data structure on several CPUs.

Protecting a data structure accessed by exceptions, interrupts, and deferrable functions

Similarly to previous cases, disabling local interrupts and acquiring a spin lock is almost always necessary to avoid race conditions. Notice that there is no need to explicitly disable deferrable functions, because they are essentially activated when terminating the execution of interrupt handlers; disabling local interrupts is therefore sufficient.

Examples of Race Condition Prevention

Kernel developers are expected to identify and solve the synchronization problems raised by interleaving kernel control paths. However, avoiding race conditions is a hard task because it requires a clear understanding of how the various components of the kernel interact. To give a feeling of what's really inside the kernel code, let's mention a few typical usages of the synchronization primitives defined in this chapter.

Reference Counters

Reference counters are widely used inside the kernel to avoid race conditions due to the concurrent allocation and releasing of a resource. A *reference counter* is just an `atomic_t` counter associated with a specific resource such as a memory page, a module, or a file. The counter is atomically increased when a kernel control path starts using the resource, and it is decreased when a kernel control path finishes using the resource. When the reference counter becomes zero, the resource is not being used, and it can be released if necessary.

The Big Kernel Lock

In earlier Linux kernel versions, a *big kernel lock* (also known as *global kernel lock*, or *BKL*) was widely used. In Linux 2.0, this lock was a relatively crude spin lock, ensuring that only one processor at a time could run in Kernel Mode. The 2.2 and 2.4 kernels were considerably more flexible and no longer relied on a single spin lock; rather, a large number of kernel data structures were protected by many different spin locks. In Linux kernel version 2.6, the big kernel lock is used to protect old code (mostly functions related to the VFS and to several filesystems).

Starting from kernel version 2.6.11, the big kernel lock is implemented by a semaphore named `kernel_sem` (in earlier 2.6 versions, the big kernel lock was implemented by means of a spin lock). The big kernel lock is slightly more sophisticated than a simple semaphore, however.

Every process descriptor includes a `lock_depth` field, which allows the same process to acquire the big kernel lock several times. Therefore, two consecutive requests for it will not hang the processor (as for normal locks). If the process has not acquired the lock, the field has the value −1; otherwise, the field value plus 1 specifies how many times the lock has been taken. The `lock_depth` field is crucial for allowing interrupt handlers, exception handlers, and deferrable functions to take the big kernel lock: without it, every asynchronous function that tries to get the big kernel lock could generate a deadlock if the current process already owns the lock.

The `lock_kernel()` and `unlock_kernel()` functions are used to get and release the big kernel lock. The former function is equivalent to:

```
depth = current->lock_depth + 1;
if (depth == 0)
    down(&kernel_sem);
current->lock_depth = depth;
```

while the latter is equivalent to:

```
if (--current->lock_depth < 0)
    up(&kernel_sem);
```

Notice that the `if` statements of the `lock_kernel()` and `unlock_kernel()` functions need not be executed atomically because `lock_depth` is not a global variable—each

CPU addresses a field of its own current process descriptor. Local interrupts inside the `if` statements do not induce race conditions either. Even if the new kernel control path invokes `lock_kernel()`, it must release the big kernel lock before terminating.

Surprisingly enough, a process holding the big kernel lock is allowed to invoke `schedule()`, thus relinquishing the CPU. The `schedule()` function, however, checks the `lock_depth` field of the process being replaced and, if its value is zero or positive, automatically releases the `kernel_sem` semaphore (see the section "The schedule() Function" in Chapter 7). Thus, no process that explicitly invokes `schedule()` can keep the big kernel lock across the process switch. The `schedule()` function, however, will reacquire the big kernel lock for the process when it will be selected again for execution.

Things are different, however, if a process that holds the big kernel lock is preempted by another process. Up to kernel version 2.6.10 this case could not occur, because acquiring a spin lock automatically disables kernel preemption. The current implementation of the big kernel lock, however, is based on a semaphore, and acquiring it does not automatically disable kernel preemption. Actually, allowing kernel preemption inside critical regions protected by the big kernel lock has been the main reason for changing its implementation. This, in turn, has beneficial effects on the response time of the system.

When a process holding the big kernel lock is preempted, `schedule()` must not release the semaphore because the process executing the code in the critical region has not voluntarily triggered a process switch, thus if the big kernel lock would be released, another process might take it and corrupt the data structures accessed by the preempted process.

To avoid the preempted process losing the big kernel lock, the `preempt_schedule_irq()` function temporarily sets the `lock_depth` field of the process to -1 (see the section "Returning from Interrupts and Exceptions" in Chapter 4). Looking at the value of this field, `schedule()` assumes that the process being replaced does not hold the `kernel_sem` semaphore and thus does not release it. As a result, the `kernel_sem` semaphore continues to be owned by the preempted process. Once this process is selected again by the scheduler, the `preempt_schedule_irq()` function restores the original value of the `lock_depth` field and lets the process resume execution in the critical section protected by the big kernel lock.

Memory Descriptor Read/Write Semaphore

Each memory descriptor of type `mm_struct` includes its own semaphore in the `mmap_sem` field (see the section "The Memory Descriptor" in Chapter 9). The semaphore protects the descriptor against race conditions that could arise because a memory descriptor can be shared among several lightweight processes.

For instance, let's suppose that the kernel must create or extend a memory region for some process; to do this, it invokes the `do_mmap()` function, which allocates a new `vm_area_struct` data structure. In doing so, the current process could be suspended if no free memory is available, and another process sharing the same memory descriptor could run. Without the semaphore, every operation of the second process that requires access to the memory descriptor (for instance, a Page Fault due to a Copy on Write) could lead to severe data corruption.

The semaphore is implemented as a read/write semaphore, because some kernel functions, such as the Page Fault exception handler (see the section "Page Fault Exception Handler" in Chapter 9), need only to scan the memory descriptors.

Slab Cache List Semaphore

The list of slab cache descriptors (see the section "Cache Descriptor" in Chapter 8) is protected by the `cache_chain_sem` semaphore, which grants an exclusive right to access and modify the list.

A race condition is possible when `kmem_cache_create()` adds a new element in the list, while `kmem_cache_shrink()` and `kmem_cache_reap()` sequentially scan the list. However, these functions are never invoked while handling an interrupt, and they can never block while accessing the list. The semaphore plays an active role both in multiprocessor systems and in uniprocessor systems with kernel preemption supported.

Inode Semaphore

As we'll see in "Inode Objects" in Chapter 12, Linux stores the information on a disk file in a memory object called an inode. The corresponding data structure includes its own semaphore in the `i_sem` field.

A huge number of race conditions can occur during filesystem handling. Indeed, each file on disk is a resource held in common for all users, because all processes may (potentially) access the file content, change its name or location, destroy or duplicate it, and so on. For example, let's suppose that a process lists the files contained in some directory. Each disk operation is potentially blocking, and therefore even in uniprocessor systems, other processes could access the same directory and modify its content while the first process is in the middle of the listing operation. Or, again, two different processes could modify the same directory at the same time. All these race conditions are avoided by protecting the directory file with the inode semaphore.

Whenever a program uses two or more semaphores, the potential for deadlock is present, because two different paths could end up waiting for each other to release a semaphore. Generally speaking, Linux has few problems with deadlocks on semaphore requests, because each kernel control path usually needs to acquire just one semaphore at a time. However, in some cases, the kernel must get two or more locks.

Inode semaphores are prone to this scenario; for instance, this occurs in the service routine in the `rename( )` system call. In this case, two different inodes are involved in the operation, so both semaphores must be taken. To avoid such deadlocks, semaphore requests are performed in predefined address order.

Timing Measurements

Countless computerized activities are driven by timing measurements, often behind the user's back. For instance, if the screen is automatically switched off after you have stopped using the computer's console, it is due to a timer that allows the kernel to keep track of how much time has elapsed since you pushed a key or moved the mouse. If you receive a warning from the system asking you to remove a set of unused files, it is the outcome of a program that identifies all user files that have not been accessed for a long time. To do these things, programs must be able to retrieve a timestamp identifying its last access time from each file. Such a timestamp must be automatically written by the kernel. More significantly, timing drives process switches along with even more visible kernel activities such as checking for time-outs.

We can distinguish two main kinds of timing measurement that must be performed by the Linux kernel:

- Keeping the current time and date so they can be returned to user programs through the time( ), ftime( ), and gettimeofday( ) APIs (see the section "The time() and gettimeofday() System Calls" later in this chapter) and used by the kernel itself as timestamps for files and network packets

- Maintaining timers—mechanisms that are able to notify the kernel (see the later section "Software Timers and Delay Functions") or a user program (see the later sections "The setitimer( ) and alarm( ) System Calls" and "System Calls for POSIX Timers") that a certain interval of time has elapsed

Timing measurements are performed by several hardware circuits based on fixed-frequency oscillators and counters. This chapter consists of four different parts. The first two sections describe the hardware devices that underly timing and give an overall picture of Linux timekeeping architecture. The following sections describe the main time-related duties of the kernel: implementing CPU time sharing, updating system time and resource usage statistics, and maintaining software timers. The last section discusses the system calls related to timing measurements and the corresponding service routines.

Clock and Timer Circuits

On the 80×86 architecture, the kernel must explicitly interact with several kinds of clocks and timer circuits. The *clock circuits* are used both to keep track of the current time of day and to make precise time measurements. The *timer circuits* are programmed by the kernel, so that they issue interrupts at a fixed, predefined frequency; such periodic interrupts are crucial for implementing the software timers used by the kernel and the user programs. We'll now briefly describe the clock and hardware circuits that can be found in IBM-compatible PCs.

Real Time Clock (RTC)

All PCs include a clock called *Real Time Clock* (*RTC*), which is independent of the CPU and all other chips.

The RTC continues to tick even when the PC is switched off, because it is energized by a small battery. The CMOS RAM and RTC are integrated in a single chip (the Motorola 146818 or an equivalent).

The RTC is capable of issuing periodic interrupts on IRQ8 at frequencies ranging between 2 Hz and 8,192 Hz. It can also be programmed to activate the IRQ8 line when the RTC reaches a specific value, thus working as an alarm clock.

Linux uses the RTC only to derive the time and date; however, it allows processes to program the RTC by acting on the */dev/rtc* device file (see Chapter 13). The kernel accesses the RTC through the `0x70` and `0x71` I/O ports. The system administrator can read and write the RTC by executing the *clock* Unix system program that acts directly on these two I/O ports.

Time Stamp Counter (TSC)

All 80×86 microprocessors include a CLK input pin, which receives the clock signal of an external oscillator. Starting with the Pentium, 80×86 microprocessors sport a counter that is increased at each clock signal. The counter is accessible through the 64-bit *Time Stamp Counter* (*TSC*) register, which can be read by means of the `rdtsc` assembly language instruction. When using this register, the kernel has to take into consideration the frequency of the clock signal: if, for instance, the clock ticks at 1 GHz, the Time Stamp Counter is increased once every nanosecond.

Linux may take advantage of this register to get much more accurate time measurements than those delivered by the Programmable Interval Timer. To do this, Linux must determine the clock signal frequency while initializing the system. In fact, because this frequency is not declared when compiling the kernel, the same kernel image may run on CPUs whose clocks may tick at any frequency.

The task of figuring out the actual frequency of a CPU is accomplished during the system's boot. The `calibrate_tsc( )` function computes the frequency by counting the number of clock signals that occur in a time interval of approximately 5 milliseconds. This time constant is produced by properly setting up one of the channels of the Programmable Interval Timer (see the next section).[*]

Programmable Interval Timer (PIT)

Besides the Real Time Clock and the Time Stamp Counter, IBM-compatible PCs include another type of time-measuring device called *Programmable Interval Timer* (*PIT*). The role of a PIT is similar to the alarm clock of a microwave oven: it makes the user aware that the cooking time interval has elapsed. Instead of ringing a bell, this device issues a special interrupt called *timer interrupt*, which notifies the kernel that one more time interval has elapsed.[†] Another difference from the alarm clock is that the PIT goes on issuing interrupts forever at some fixed frequency established by the kernel. Each IBM-compatible PC includes at least one PIT, which is usually implemented by an 8254 CMOS chip using the `0x40–0x43` I/O ports.

As we'll see in detail in the next paragraphs, Linux programs the PIT of IBM-compatible PCs to issue timer interrupts on the IRQ0 at a (roughly) 1000-Hz frequency—that is, once every 1 millisecond. This time interval is called a *tick*, and its length in nanoseconds is stored in the `tick_nsec` variable. On a PC, `tick_nsec` is initialized to 999,848 nanoseconds (yielding a clock signal frequency of about 1000.15 Hz), but its value may be automatically adjusted by the kernel if the computer is synchronized with an external clock (see the later section "The adjtimex( ) System Call"). The ticks beat time for all activities in the system; in some sense, they are like the ticks sounded by a metronome while a musician is rehearsing.

Generally speaking, shorter ticks result in higher resolution timers, which help with smoother multimedia playback and faster response time when performing synchronous I/O multiplexing (`poll()` and `select()` system calls). This is a trade-off however: shorter ticks require the CPU to spend a larger fraction of its time in Kernel Mode—that is, a smaller fraction of time in User Mode. As a consequence, user programs run slower.

The frequency of timer interrupts depends on the hardware architecture. The slower machines have a tick of roughly 10 milliseconds (100 timer interrupts per second), while the faster ones have a tick of roughly 1 millisecond (1000 or 1024 timer interrupts per second).

---

[*] To avoid losing significant digits in the integer divisions, `calibrate_tsc( )` returns the duration, in microseconds, of a clock tick multiplied by 2^{32}.

[†] The PIT is also used to drive an audio amplifier connected to the computer's internal speaker.

A few macros in the Linux code yield some constants that determine the frequency of timer interrupts. These are discussed in the following list.

- HZ yields the approximate number of timer interrupts per second—that is, their frequency. This value is set to 1000 for IBM PCs.
- CLOCK_TICK_RATE yields the value 1,193,182, which is the 8254 chip's internal oscillator frequency.
- LATCH yields the ratio between CLOCK_TICK_RATE and HZ, rounded to the nearest integer. It is used to program the PIT.

The PIT is initialized by setup_pit_timer( ) as follows:

```
spin_lock_irqsave(&i8253_lock, flags);
outb_p(0x34,0x43);
udelay(10);
outb_p(LATCH & 0xff, 0x40);
udelay(10);
outb(LATCH >> 8, 0x40);
spin_unlock_irqrestore(&i8253_lock, flags);
```

The outb( ) C function is equivalent to the outb assembly language instruction: it copies the first operand into the I/O port specified as the second operand. The outb_p( ) function is similar to outb( ), except that it introduces a pause by executing a no-op instruction to keep the hardware from getting confused. The udelay() macro introduces a further small delay (see the later section "Delay Functions"). The first outb_p( ) invocation is a command to the PIT to issue interrupts at a new rate. The next two outb_p( ) and outb( ) invocations supply the new interrupt rate to the device. The 16-bit LATCH constant is sent to the 8-bit 0x40 I/O port of the device as two consecutive bytes. As a result, the PIT issues timer interrupts at a (roughly) 1000-Hz frequency (that is, once every 1 ms).

CPU Local Timer

The local APIC present in recent 80×86 microprocessors (see the section "Interrupts and Exceptions" in Chapter 4) provides yet another time-measuring device: the *CPU local timer*.

The CPU local timer is a device similar to the Programmable Interval Timer just described that can issue one-shot or periodic interrupts. There are, however, a few differences:

- The APIC's timer counter is 32 bits long, while the PIT's timer counter is 16 bits long; therefore, the local timer can be programmed to issue interrupts at very low frequencies (the counter stores the number of ticks that must elapse before the interrupt is issued).
- The local APIC timer sends an interrupt only to its processor, while the PIT raises a global interrupt, which may be handled by any CPU in the system.

- The APIC's timer is based on the bus clock signal (or the APIC bus signal, in older machines). It can be programmed in such a way to decrease the timer counter every 1, 2, 4, 8, 16, 32, 64, or 128 bus clock signals. Conversely, the PIT, which makes use of its own clock signals, can be programmed in a more flexible way.

High Precision Event Timer (HPET)

The *High Precision Event Timer* (*HPET*) is a new timer chip developed jointly by Intel and Microsoft. Although HPETs are not yet very common in end-user machines, Linux 2.6 already supports them, so we'll spend a few words describing their characteristics.

The HPET provides a number of hardware timers that can be exploited by the kernel. Basically, the chip includes up to eight 32-bit or 64-bit independent *counters*. Each counter is driven by its own clock signal, whose frequency must be at least 10 MHz; therefore, the counter is increased at least once in 100 nanoseconds. Any counter is associated with at most 32 *timers*, each of which is composed by a *comparator* and a *match register*. The comparator is a circuit that checks the value in the counter against the value in the match register, and raises a hardware interrupt if a match is found. Some of the timers can be enabled to generate a periodic interrupt.

The HPET chip can be programmed through registers mapped into memory space (much like the I/O APIC). The BIOS establishes the mapping during the bootstrapping phase and reports to the operating system kernel its initial memory address. The HPET registers allow the kernel to read and write the values of the counters and of the match registers, to program one-shot interrupts, and to enable or disable periodic interrupts on the timers that support them.

The next generation of motherboards will likely support both the HPET and the 8254 PIT; in some future time, however, the HPET is expected to completely replace the PIT.

ACPI Power Management Timer

The *ACPI Power Management Timer* (or *ACPI PMT*) is yet another clock device included in almost all ACPI-based motherboards. Its clock signal has a fixed frequency of roughly 3.58 MHz. The device is actually a simple counter increased at each clock tick; to read the current value of the counter, the kernel accesses an I/O port whose address is determined by the BIOS during the initialization phase (see Appendix A).

The ACPI Power Management Timer is preferable to the TSC if the operating system or the BIOS may dynamically lower the frequency or voltage of the CPU to save battery power. When this happens, the frequency of the TSC changes—thus causing time warps and others unpleasant effects—while the frequency of the ACPI PMT

does not. On the other hand, the high-frequency of the TSC counter is quite handy for measuring very small time intervals.

However, if an HPET device is present, it should always be preferred to the other circuits because of its richer architecture. Table 6-2 later in this chapter illustrates how Linux takes advantage of the available timing circuits.

Now that we understand what the hardware timers are, we may discuss how the Linux kernel exploits them to conduct all activities of the system.

The Linux Timekeeping Architecture

Linux must carry on several time-related activities. For instance, the kernel periodically:

- Updates the time elapsed since system startup.
- Updates the time and date.
- Determines, for every CPU, how long the current process has been running, and preempts it if it has exceeded the time allocated to it. The allocation of time slots (also called "quanta") is discussed in Chapter 7.
- Updates resource usage statistics.
- Checks whether the interval of time associated with each software timer (see the later section "Software Timers and Delay Functions") has elapsed.

Linux's *timekeeping architecture* is the set of kernel data structures and functions related to the flow of time. Actually, 80×86-based multiprocessor machines have a timekeeping architecture that is slightly different from the timekeeping architecture of uniprocessor machines:

- In a uniprocessor system, all time-keeping activities are triggered by interrupts raised by the global timer (either the Programmable Interval Timer or the High Precision Event Timer).
- In a multiprocessor system, all general activities (such as handling of software timers) are triggered by the interrupts raised by the global timer, while CPU-specific activities (such as monitoring the execution time of the currently running process) are triggered by the interrupts raised by the local APIC timer.

Unfortunately, the distinction between the two cases is somewhat blurred. For instance, some early SMP systems based on Intel 80486 processors didn't have local APICs. Even nowadays, there are SMP motherboards so buggy that local timer interrupts are not usable at all. In these cases, the SMP kernel must resort to the UP timekeeping architecture. On the other hand, recent uniprocessor systems feature one local APIC, so the UP kernel often makes use of the SMP timekeeping architecture. However, to simplify our description, we won't discuss these hybrid cases and will stick to the two "pure" timekeeping architectures.

Linux's timekeeping architecture depends also on the availability of the Time Stamp Counter (TSC), of the ACPI Power Management Timer, and of the High Precision Event Timer (HPET). The kernel uses two basic timekeeping functions: one to keep the current time up-to-date and another to count the number of nanoseconds that have elapsed within the current second. There are different ways to get the last value. Some methods are more precise and are available if the CPU has a Time Stamp Counter or a HPET; a less-precise method is used in the opposite case (see the later section "The time( ) and gettimeofday( ) System Calls").

Data Structures of the Timekeeping Architecture

The timekeeping architecture of Linux 2.6 makes use of a large number of data structures. As usual, we will describe the most important variables by referring to the 80×86 architecture.

The timer object

In order to handle the possible timer sources in a uniform way, the kernel makes use of a "timer object," which is a descriptor of type `timer_opts` consisting of the timer name and of four standard methods shown in Table 6-1.

Table 6-1. The fields of the timer_opts data structure

Field name	Description
name	A string identifying the timer source
mark_offset	Records the exact time of the last tick; it is invoked by the timer interrupt handler
get_offset	Returns the time elapsed since the last tick
monotonic_clock	Returns the number of nanoseconds since the kernel initialization
delay	Waits for a given number of "loops" (see the later section "Delay Functions")

The most important methods of the timer object are `mark_offset` and `get_offset`. The `mark_offset` method is invoked by the timer interrupt handler, and records in a suitable data structure the exact time at which the tick occurred. Using the saved value, the `get_offset` method computes the time in microseconds elapsed since the last timer interrupt (tick). Thanks to these two methods, Linux timekeeping architecture achieves a sub-tick resolution—that is, the kernel is able to determine the current time with a precision much higher than the tick duration. This operation is called *time interpolation*.

The `cur_timer` variable stores the address of the timer object corresponding to the "best" timer source available in the system. Initially, `cur_timer` points to `timer_none`, which is the object corresponding to a dummy timer source used when the kernel is being initialized. During kernel initialization, the `select_timer( )` function sets `cur_timer` to the address of the appropriate timer object. Table 6-2 shows the most common timer objects used in the 80×86 architecture, in order of preference. As you see,

select_timer() selects the HPET, if available; otherwise, it selects the ACPI Power Management Timer, if available, or the TSC. As the last resort, select_timer() selects the always-present PIT. The "Time interpolation" column lists the timer sources used by the mark_offset and get_offset methods of the timer object; the "Delay" column lists the timer sources used by the delay method.

Table 6-2. Typical timer objects of the 80×86 architecture, in order of preference

Timer object name	Description	Time interpolation	Delay
timer_hpet	High Precision Event Timer (HPET)	HPET	HPET
timer_pmtmr	ACPI Power Management Timer (ACPI PMT)	ACPI PMT	TSC
timer_tsc	Time Stamp Counter (TSC)	TSC	TSC
timer_pit	Programmable Interval Timer (PIT)	PIT	Tight loop
timer_none	Generic dummy timer source (used during kernel initialization)	(none)	Tight loop

Notice that local APIC timers do not have a corresponding timer object. The reason is that local APIC timers are used only to generate periodic interrupts and are never used to achieve sub-tick resolution.

The jiffies variable

The jiffies variable is a counter that stores the number of elapsed ticks since the system was started. It is increased by one when a timer interrupt occurs—that is, on every tick. In the 80×86 architecture, jiffies is a 32-bit variable, therefore it wraps around in approximately 50 days—a relatively short time interval for a Linux server. However, the kernel handles cleanly the overflow of jiffies thanks to the time_after, time_after_eq, time_before, and time_before_eq macros: they yield the correct value even if a wraparound occurred.

You might suppose that jiffies is initialized to zero at system startup. Actually, this is not the case: jiffies is initialized to 0xfffb6c20, which corresponds to the 32-bit signed value −300,000; therefore, the counter will overflow five minutes after the system boot. This is done on purpose, so that buggy kernel code that does not check for the overflow of jiffies shows up very soon in the developing phase and does not pass unnoticed in stable kernels.

In a few cases, however, the kernel needs the real number of system ticks elapsed since the system boot, regardless of the overflows of jiffies. Therefore, in the 80×86 architecture the jiffies variable is equated by the linker to the 32 less significant bits of a 64-bit counter called jiffies_64. With a tick of 1 millisecond, the jiffies_64 variable wraps around in several hundreds of millions of years, thus we can safely assume that it never overflows.

You might wonder why jiffies has not been directly declared as a 64-bit unsigned long long integer on the 80×86 architecture. The answer is that accesses to 64-bit vari-

ables in 32-bit architectures cannot be done atomically. Therefore, every read operation on the whole 64 bits requires some synchronization technique to ensure that the counter is not updated while the two 32-bit half-counters are read; as a consequence, every 64-bit read operation is significantly slower than a 32-bit read operation.

The get_jiffies_64( ) function reads the value of jiffies_64 and returns its value:

```
unsigned long long get_jiffies_64(void)
{
    unsigned long seq;
    unsigned long long ret;
    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xime_lock, seq));
    return ret;
}
```

The 64-bit read operation is protected by the xtime_lock seqlock (see the section "Seqlocks" in Chapter 5): the function keeps reading the jiffies_64 variable until it knows for sure that it has not been concurrently updated by another kernel control path.

Conversely, the critical region increasing the jiffies_64 variable must be protected by means of write_seqlock(&xtime_lock) and write_sequnlock(&xtime_lock). Notice that the ++jiffies_64 instruction also increases the 32-bit jiffies variable, because the latter corresponds to the lower half of jiffies_64.

The xtime variable

The xtime variable stores the current time and date; it is a structure of type timespec having two fields:

tv_sec
    Stores the number of seconds that have elapsed since midnight of January 1, 1970 (UTC)

tv_nsec
    Stores the number of nanoseconds that have elapsed within the last second (its value ranges between 0 and 999,999,999)

The xtime variable is usually updated once in a tick—that is, roughly 1000 times per second. As we'll see in the later section "System Calls Related to Timing Measurements," user programs get the current time and date from the xtime variable. The kernel also often refers to it, for instance, when updating inode timestamps (see the section "File Descriptor and Inode" in Chapter 1).

The xtime_lock seqlock avoids the race conditions that could occur due to concurrent accesses to the xtime variable. Remember that xtime_lock also protects the jiffies_64 variable; in general, this seqlock is used to define several critical regions of the timekeeping architecture.

Timekeeping Architecture in Uniprocessor Systems

In a uniprocessor system, all time-related activities are triggered by the interrupts raised by the Programmable Interval Timer on IRQ line 0. As usual, in Linux, some of these activities are executed as soon as possible right after the interrupt is raised, while the remaining activities are carried on by deferrable functions (see the later section "Dynamic Timers").

Initialization phase

During kernel initialization, the `time_init()` function is invoked to set up the time-keeping architecture. It usually[*] performs the following operations:

1. Initializes the `xtime` variable. The number of seconds elapsed since the midnight of January 1, 1970 is read from the Real Time Clock by means of the `get_cmos_time()` function. The `tv_nsec` field of `xtime` is set, so that the forthcoming over-flow of the `jiffies` variable will coincide with an increment of the `tv_sec` field—that is, it will fall on a second boundary.

2. Initializes the `wall_to_monotonic` variable. This variable is of the same type `timespec` as `xtime`, and it essentially stores the number of seconds and nanoseconds to be added to `xtime` in order to get a monotonic (ever increasing) flow of time. In fact, both leap seconds and synchronization with external clocks might suddenly change the `tv_sec` and `tv_nsec` fields of `xtime` so that they are no longer monotonically increased. As we'll see in the later section "System Calls for POSIX Timers," sometimes the kernel needs a truly monotonic time source.

3. If the kernel supports HPET, it invokes the `hpet_enable()` function to determine whether the ACPI firmware has probed the chip and mapped its registers in the memory address space. In the affirmative case, `hpet_enable()` programs the first timer of the HPET chip so that it raises the IRQ0 interrupt 1000 times per second. Otherwise, if the HPET chip is not available, the kernel will use the PIT: the chip has already been programmed by the `init_IRQ()` function to raise 1000 timer interrupts per second, as described in the earlier section "Programmable Interval Timer (PIT)."

4. Invokes `select_timer()` to select the best timer source available in the system, and sets the `cur_timer` variable to the address of the corresponding timer object.

---

[*] The `time_init()` function is executed before `mem_init()`, which initializes the memory data structures. Unfortunately, the HPET registers are memory mapped, therefore initialization of the HPET chip has to be done after the execution of `mem_init()`. Linux 2.6 adopts a cumbersome solution: if the kernel supports the HPET chip, the `time_init()` function limits itself to trigger the activation of the `hpet_time_init()` function. The latter function is executed after `mem_init()` and performs the operations described in this section.

5. Invokes `setup_irq(0,&irq0)` to set up the interrupt gate corresponding to IRQ0—the line associated with the system timer interrupt source (PIT or HPET).The `irq0` variable is statically defined as:

```
struct irqaction irq0 = { timer_interrupt, SA_INTERRUPT, 0,
                          "timer", NULL, NULL };
```

From now on, the `timer_interrupt()` function will be invoked once every tick with interrupts disabled, because the `status` field of IRQ0's main descriptor has the `SA_INTERRUPT` flag set.

The timer interrupt handler

The `timer_interrupt()` function is the interrupt service routine (ISR) of the PIT or of the HPET; it performs the following steps:

1. Protects the time-related kernel variables by issuing a `write_seqlock()` on the `xtime_lock` seqlock (see the section "Seqlocks" in Chapter 5).

2. Executes the `mark_offset` method of the `cur_timer` timer object. As explained in the earlier section "Data Structures of the Timekeeping Architecture," there are four possible cases:

   a. `cur_timer` points to the `timer_hpet` object: in this case, the HPET chip is the source of timer interrupts. The `mark_offset` method checks that no timer interrupt has been lost since the last tick; in this unlikely case, it updates `jiffies_64` accordingly. Next, the method records the current value of the periodic HPET counter.

   b. `cur_timer` points to the `timer_pmtmr` object: in this case, the PIT chip is the source of timer interrupts, but the kernel uses the APIC Power Management Timer to measure time with a finer resolution. The `mark_offset` method checks that no timer interrupt has been lost since the last tick and updates `jiffies_64` if necessary. Then, it records the current value of the APIC Power Management Timer counter.

   c. `cur_timer` points to the `timer_tsc` object: in this case, the PIT chip is the source of timer interrupts, but the kernel uses the Time Stamp Counter to measure time with a finer resolution. The `mark_offset` method performs the same operations as in the previous case: it checks that no timer interrupt has been lost since the last tick and updates `jiffies_64` if necessary. Then, it records the current value of the TSC counter.

   d. `cur_timer` points to the `timer_pit` object: in this case, the PIT chip is the source of timer interrupts, and there is no other timer circuit. The `mark_offset` method does nothing.

3. Invokes the `do_timer_interrupt()` function, which in turn performs the following actions:

   a. Increases by one the value of `jiffies_64`. Notice that this can be done safely, because the kernel control path still holds the `xtime_lock` seqlock for writing.

b. Invokes the `update_times()` function to update the system date and time and to compute the current system load; these activities are discussed later in the sections "Updating the Time and Date" and "Updating System Statistics."

c. Invokes the `update_process_times()` function to perform several time-related accounting operations for the local CPU (see the section "Updating Local CPU Statistics" later in this chapter).

d. Invokes the `profile_tick()` function (see the section "Profiling the Kernel Code" later in this chapter).

e. If the system clock is synchronized with an external clock (an `adjtimex()` system call has been previously issued), invokes the `set_rtc_mmss()` function once every 660 seconds (every 11 minutes) to adjust the Real Time Clock. This feature helps systems on a network synchronize their clocks (see the later section "The adjtimex() System Call").

4. Releases the `xtime_lock` seqlock by invoking `write_sequnlock()`.

5. Returns the value 1 to notify that the interrupt has been effectively handled (see the section "I/O Interrupt Handling" in Chapter 4).

Timekeeping Architecture in Multiprocessor Systems

Multiprocessor systems can rely on two different sources of timer interrupts: those raised by the Programmable Interval Timer or the High Precision Event Timer, and those raised by the CPU local timers.

In Linux 2.6, global timer interrupts—raised by the PIT or the HPET—signal activities not related to a specific CPU, such as handling of software timers and keeping the system time up-to-date. Conversely, a CPU local timer interrupt signals timekeeping activities related to the local CPU, such as monitoring how long the current process has been running and updating the resource usage statistics.

Initialization phase

The global timer interrupt handler is initialized by the `time_init()` function, which has already been described in the earlier section "Timekeeping Architecture in Uniprocessor Systems."

The Linux kernel reserves the interrupt vector 239 (`0xef`) for local timer interrupts (see Table 4-2 in Chapter 4). During kernel initialization, the `apic_intr_init()` function sets up the IDT's interrupt gate corresponding to vector 239 with the address of the low-level interrupt handler `apic_timer_interrupt()`. Moreover, each APIC has to be told how often to generate a local time interrupt. The `calibrate_APIC_clock()` function computes how many bus clock signals are received by the local APIC of the booting CPU during a tick (1 ms). This exact value is then used to program the local APICs

in such a way to generate one local timer interrupt every tick. This is done by the `setup_APIC_timer()` function, which is executed once for every CPU in the system.

All local APIC timers are synchronized because they are based on the common bus clock signal. This means that the value computed by `calibrate_APIC_clock()` for the boot CPU is also good for the other CPUs in the system.

The global timer interrupt handler

The SMP version of the `timer_interrupt()` handler differs from the UP version in a few points:

- The `do_timer_interrupt()` function, invoked by `timer_interrupt()`, writes into a port of the I/O APIC chip to acknowledge the timer IRQ.
- The `update_process_times()` function is not invoked, because this function performs actions related to a specific CPU.
- The `profile_tick()` function is not invoked, because this function also performs actions related to a specific CPU.

The local timer interrupt handler

This handler performs the timekeeping activities related to a specific CPU in the system, namely profiling the kernel code and checking how long the current process has been running on a given CPU.

The `apic_timer_interrupt()` assembly language function is equivalent to the following code:

```
apic_timer_interrupt:
    pushl $(239-256)
    SAVE_ALL
    movl %esp, %eax
    call smp_apic_timer_interrupt
    jmp ret_from_intr
```

As you can see, the low-level handler is very similar to the other low-level interrupt handlers already described in Chapter 4. The high-level interrupt handler called `smp_apic_timer_interrupt()` executes the following steps:

1. Gets the CPU logical number (say, n).
2. Increases the `apic_timer_irqs` field of the nth entry of the `irq_stat` array (see the section "Checking the NMI Watchdogs" later in this chapter).
3. Acknowledges the interrupt on the local APIC.
4. Calls the `irq_enter()` function (see the section "The do_IRQ() function" in Chapter 4).
5. Invokes the `smp_local_timer_interrupt()` function.
6. Calls the `irq_exit()` function.

The `smp_local_timer_interrupt()` function executes the per-CPU timekeeping activities. Actually, it performs the following main steps:

1. Invokes the `profile_tick()` function (see the section "Profiling the Kernel Code" later in this chapter).

2. Invokes the `update_process_times()` function to check how long the current process has been running and to update some local CPU statistics (see the section "Updating Local CPU Statistics" later in this chapter).

The system administrator can change the sample frequency of the kernel code profiler by writing into the */proc/profile* file.To carry out the change, the kernel modifies the frequency at which local timer interrupts are generated. However, the `smp_local_timer_interrupt()` function keeps invoking the `update_process_times()` function exactly once every tick.

Updating the Time and Date

User programs get the current time and date from the `xtime` variable. The kernel must periodically update this variable, so that its value is always reasonably accurate.

The `update_times()` function, which is invoked by the global timer interrupt handler, updates the value of the `xtime` variable as follows:

```
void update_times(void)
{
    unsigned long ticks;
    ticks = jiffies - wall_jiffies;
    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    calc_load(ticks);
}
```

We recall from the previous description of the timer interrupt handler that when the code of this function is executed, the `xtime_lock` seqlock has already been acquired for writing.

The `wall_jiffies` variable stores the time of the last update of the `xtime` variable. Observe that the value of `wall_jiffies` can be smaller than `jiffies-1`, since a few timer interrupts can be lost, for instance when interrupts remain disabled for a long period of time; in other words, the kernel does not necessarily update the `xtime` variable at every tick. However, no tick is definitively lost, and in the long run, `xtime` stores the correct system time. The check for lost timer interrupts is done in the `mark_offset` method of `cur_timer`; see the earlier section "Timekeeping Architecture in Uniprocessor Systems."

The update_wall_time( ) function invokes the update_wall_time_one_tick( ) function ticks consecutive times; normally, each invocation adds 1,000,000 to the xtime.tv_nsec field. If the value of xtime.tv_nsec becomes greater than 999,999,999, the update_wall_time( ) function also updates the tv_sec field of xtime. If an adjtimex( ) system call has been issued, for reasons explained in the section "The adjtimex( ) System Call" later in this chapter, the function might tune the value 1,000,000 slightly so the clock speeds up or slows down a little.

The calc_load( ) function is described in the section "Keeping Track of System Load" later in this chapter.

Updating System Statistics

The kernel, among the other time-related duties, must periodically collect various data used for:

- Checking the CPU resource limit of the running processes
- Updating statistics about the local CPU workload
- Computing the average system load
- Profiling the kernel code

Updating Local CPU Statistics

We have mentioned that the update_process_times( ) function is invoked—either by the global timer interrupt handler on uniprocessor systems or by the local timer interrupt handler in multiprocessor systems—to update some kernel statistics. This function performs the following steps:

1. Checks how long the current process has been running. Depending on whether the current process was running in User Mode or in Kernel Mode when the timer interrupt occurred, invokes either account_user_time( ) or account_system_time( ). Each of these functions performs essentially the following steps:

   a. Updates either the utime field (ticks spent in User Mode) or the stime field (ticks spent in Kernel Mode) of the current process descriptor. Two additional fields called cutime and cstime are provided in the process descriptor to count the number of CPU ticks spent by the process children in User Mode and Kernel Mode, respectively. For reasons of efficiency, these fields are not updated by update_process_times( ), but rather when the parent process queries the state of one of its children (see the section "Destroying Processes" in Chapter 3).

   b. Checks whether the total CPU time limit has been reached; if so, sends SIGXCPU and SIGKILL signals to current. The section "Process Resource Limits" in Chapter 3 describes how the limit is controlled by the signal->rlim[RLIMIT_CPU].rlim_cur field of each process descriptor.

c. Invokes `account_it_virt()` and `account_it_prof()` to check the process timers (see the section "The setitimer() and alarm() System Calls" later in this chapter).

d. Updates some kernel statistics stored in the `kstat` per-CPU variable.

2. Invokes `raise_softirq()` to activate the `TIMER_SOFTIRQ` tasklet on the local CPU (see the section "Software Timers and Delay Functions" later in this chapter).

3. If some old version of an RCU-protected data structure has to be reclaimed, checks whether the local CPU has gone through a quiescent state and invokes `tasklet_schedule()` to activate the `rcu_tasklet` tasklet of the local CPU (see the section "Read-Copy Update (RCU)" in Chapter 5).

4. Invokes the `scheduler_tick()` function, which decreases the time slice counter of the current process, and checks whether its quantum is exhausted. We'll discuss in depth these operations in the section "The scheduler_tick() Function" in Chapter 7.

Keeping Track of System Load

Every Unix kernel keeps track of how much CPU activity is being carried on by the system. These statistics are used by various administration utilities such as `top`. A user who enters the `uptime` command sees the statistics as the "load average" relative to the last minute, the last 5 minutes, and the last 15 minutes. On a uniprocessor system, a value of 0 means that there are no active processes (besides the *swapper* process 0) to run, while a value of 1 means that the CPU is 100 percent busy with a single process, and values greater than 1 mean that the CPU is shared among several active processes.[*]

At every tick, `update_times()` invokes the `calc_load()` function, which counts the number of processes in the `TASK_RUNNING` or `TASK_UNINTERRUPTIBLE` state and uses this number to update the average system load.

Profiling the Kernel Code

Linux includes a minimalist code profiler called *readprofile* used by Linux developers to discover where the kernel spends its time in Kernel Mode. The profiler identifies the *hot spots* of the kernel—the most frequently executed fragments of kernel code. Identifying the kernel hot spots is very important, because they may point out kernel functions that should be further optimized.

---

[*] Linux includes in the load average all processes that are in the `TASK_RUNNING` and `TASK_UNINTERRUPTIBLE` states. However, under normal conditions, there are few `TASK_UNINTERRUPTIBLE` processes, so a high load usually means that the CPU is busy.

The profiler is based on a simple Monte Carlo algorithm: at every timer interrupt occurrence, the kernel determines whether the interrupt occurred in Kernel Mode; if so, the kernel fetches the value of the eip register before the interruption from the stack and uses it to discover what the kernel was doing before the interrupt. In the long run, the samples accumulate on the hot spots.

The profile_tick() function collects the data for the code profiler. It is invoked either by the do_timer_interrupt() function in uniprocessor systems (by the global timer interrupt handler) or by the smp_local_timer_interrupt() function in multi-processor systems (by the local timer interrupt handler).

To enable the code profiler, the Linux kernel must be booted by passing as a parameter the string profile=N, where 2^N denotes the size of the code fragments to be profiled. The collected data can be read from the */proc/profile* file. The counters are reset by writing in the same file; in multiprocessor systems, writing into the file can also change the sample frequency (see the earlier section "Timekeeping Architecture in Multiprocessor Systems"). However, kernel developers do not usually access */proc/profile* directly; instead, they use the *readprofile* system command.

The Linux 2.6 kernel includes yet another profiler called *oprofile*. Besides being more flexible and customizable than *readprofile*, *oprofile* can be used to discover hot spots in kernel code, User Mode applications, and system libraries. When *oprofile* is being used, profile_tick() invokes the timer_notify() function to collect the data used by this new profiler.

Checking the NMI Watchdogs

In multiprocessor systems, Linux offers yet another feature to kernel developers: a *watchdog system*, which might be quite useful to detect kernel bugs that cause a system freeze. To activate such a watchdog, the kernel must be booted with the nmi_watchdog parameter.

The watchdog is based on a clever hardware feature of local and I/O APICs: they can generate periodic NMI interrupts on every CPU. Because NMI interrupts are not masked by the cli assembly language instruction, the watchdog can detect deadlocks even when interrupts are disabled.

As a consequence, once every tick, all CPUs, regardless of what they are doing, start executing the NMI interrupt handler; in turn, the handler invokes do_nmi(). This function gets the logical number n of the CPU, and then checks the apic_timer_irqs field of the nth entry of irq_stat (see Table 4-8 in Chapter 4). If the CPU is working properly, the value must be different from the value read at the previous NMI interrupt. When the CPU is running properly, the nth entry of the apic_timer_irqs field is increased by the local timer interrupt handler (see the earlier section "The local timer interrupt handler"); if the counter is not increased, the local timer interrupt handler has not been executed in a whole tick. Not a good thing, you know.

When the NMI interrupt handler detects a CPU freeze, it rings all the bells: it logs scary messages in the system logfiles, dumps the contents of the CPU registers and of the kernel stack (kernel oops), and finally kills the current process. This gives kernel developers a chance to discover what's gone wrong.

Software Timers and Delay Functions

A *timer* is a software facility that allows functions to be invoked at some future moment, after a given time interval has elapsed; a *time-out* denotes a moment at which the time interval associated with a timer has elapsed.

Timers are widely used both by the kernel and by processes. Most device drivers use timers to detect anomalous conditions—floppy disk drivers, for instance, use timers to switch off the device motor after the floppy has not been accessed for a while, and parallel printer drivers use them to detect erroneous printer conditions.

Timers are also used quite often by programmers to force the execution of specific functions at some future time (see the later section "The setitimer( ) and alarm( ) System Calls").

Implementing a timer is relatively easy. Each timer contains a field that indicates how far in the future the timer should expire. This field is initially calculated by adding the right number of ticks to the current value of `jiffies`. The field does not change. Every time the kernel checks a timer, it compares the expiration field to the value of `jiffies` at the current moment, and the timer expires when `jiffies` is greater than or equal to the stored value.

Linux considers two types of timers called *dynamic timers* and *interval timers*. The first type is used by the kernel, while interval timers may be created by processes in User Mode.

One word of caution about Linux timers: since checking for timer functions is always done by deferrable functions that may be executed a long time after they have been activated, the kernel cannot ensure that timer functions will start right at their expiration times. It can only ensure that they are executed either at the proper time or after with a delay of up to a few hundreds of milliseconds. For this reason, timers are not appropriate for real-time applications in which expiration times must be strictly enforced.

Besides software timers, the kernel also makes use of *delay functions*, which execute a tight instruction loop until a given time interval elapses. We will discuss them in the later section "Delay Functions."

Dynamic Timers

Dynamic timers may be dynamically created and destroyed. No limit is placed on the number of currently active dynamic timers.

A dynamic timer is stored in the following `timer_list` structure:

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    spinlock_t lock;
    unsigned long magic;
    void (*function)(unsigned long);
    unsigned long data;
    tvec_base_t *base;
};
```

The `function` field contains the address of the function to be executed when the timer expires. The `data` field specifies a parameter to be passed to this timer function. Thanks to the `data` field, it is possible to define a single general-purpose function that handles the time-outs of several device drivers; the `data` field could store the device ID or other meaningful data that could be used by the function to differentiate the device.

The `expires` field specifies when the timer expires; the time is expressed as the number of ticks that have elapsed since the system started up. All timers that have an `expires` value smaller than or equal to the value of `jiffies` are considered to be expired or decayed.

The `entry` field is used to insert the software timer into one of the doubly linked circular lists that group together the timers according to the value of their `expires` field. The algorithm that uses these lists is described later in this chapter.

To create and activate a dynamic timer, the kernel must:

1. Create, if necessary, a new `timer_list` object—for example, `t`. This can be done in several ways by:
   - Defining a static global variable in the code.
   - Defining a local variable inside a function; in this case, the object is stored on the Kernel Mode stack.
   - Including the object in a dynamically allocated descriptor.
2. Initialize the object by invoking the `init_timer(&t)` function. This essentially sets the `t.base` pointer field to `NULL` and sets the `t.lock` spin lock to "open."
3. Load the `function` field with the address of the function to be activated when the timer decays. If required, load the `data` field with a parameter value to be passed to the function.
4. If the dynamic timer is not already inserted in a list, assign a proper value to the `expires` field and invoke the `add_timer(&t)` function to insert the `t` element in the proper list.
5. Otherwise, if the dynamic timer is already inserted in a list, update the `expires` field by invoking the `mod_timer()` function, which also takes care of moving the object into the proper list (discussed next).

Once the timer has decayed, the kernel automatically removes the `t` element from its list. Sometimes, however, a process should explicitly remove a timer from its list using the `del_timer()`, `del_timer_sync()`, or `del_singleshot_timer_sync()` functions. Indeed, a sleeping process may be woken up before the time-out is over; in this case, the process may choose to destroy the timer. Invoking `del_timer()` on a timer already removed from a list does no harm, so removing the timer within the timer function is considered a good practice.

In Linux 2.6, a dynamic timer is bound to the CPU that activated it—that is, the timer function will always run on the same CPU that first executed the `add_timer()` or later the `mod_timer()` function. The `del_timer()` and companion functions, however, can deactivate every dynamic timer, even if it is not bound to the local CPU.

Dynamic timers and race conditions

Being asynchronously activated, dynamic timers are prone to race conditions. For instance, consider a dynamic timer whose function acts on a discardable resource (e.g., a kernel module or a file data structure). Releasing the resource without stopping the timer may lead to data corruption if the timer function got activated when the resource no longer exists. Thus, a rule of thumb is to stop the timer *before* releasing the resource:

```
...
del_timer(&t);
X_Release_Resources();
...
```

In multiprocessor systems, however, this code is not safe because the timer function might already be running on another CPU when `del_timer()` is invoked. As a result, resources may be released while the timer function is still acting on them. To avoid this kind of race condition, the kernel offers the `del_timer_sync()` function. It removes the timer from the list, and then it checks whether the timer function is executed on another CPU; in such a case, `del_timer_sync()` waits until the timer function terminates.

The `del_timer_sync()` function is rather complex and slow, because it has to carefully take into consideration the case in which the timer function reactivates itself. If the kernel developer knows that the timer function never reactivates the timer, she can use the simpler and faster `del_singleshot_timer_sync()` function to deactivate a timer and wait until the timer function terminates.

Other types of race conditions exist, of course. For instance, the right way to modify the `expires` field of an already activated timer consists of using `mod_timer()`, rather than deleting the timer and re-creating it thereafter. In the latter approach, two kernel control paths that want to modify the `expires` field of the same timer may mix each other up badly. The implementation of the timer functions is made SMP-safe by means of the `lock` spin lock included in every `timer_list` object: every time the kernel must access a dynamic timer, it disables the interrupts and acquires this spin lock.

Data structures for dynamic timers

Choosing the proper data structure to implement dynamic timers is not easy. Stringing together all timers in a single list would degrade system performance, because scanning a long list of timers at every tick is costly. On the other hand, maintaining a sorted list would not be much more efficient, because the insertion and deletion operations would also be costly.

The adopted solution is based on a clever data structure that partitions the expires values into blocks of ticks and allows dynamic timers to percolate efficiently from lists with larger expires values to lists with smaller ones. Moreover, in multiprocessor systems the set of active dynamic timers is split among the various CPUs.

The main data structure for dynamic timers is a per-CPU variable (see the section "Per-CPU Variables" in Chapter 5) named tvec_bases: it includes NR_CPUS elements, one for each CPU in the system. Each element is a tvec_base_t structure, which includes all data needed to handle the dynamic timers bound to the corresponding CPU:

```
typedef struct tvec_t_base_s {
    spinlock_t lock;
    unsigned long timer_jiffies;
    struct timer_list *running_timer;
    tvec_root_t tv1;
    tvec_t tv2;
    tvec_t tv3;
    tvec_t tv4;
    tvec_t tv5;
} tvec_base_t;
```

The tv1 field is a structure of type tvec_root_t, which includes a vec array of 256 list_head elements—that is, lists of dynamic timers. It contains all dynamic timers, if any, that will decay within the next 255 ticks.

The tv2, tv3, and tv4 fields are structures of type tvec_t consisting of a vec array of 64 list_head elements. These lists contain all dynamic timers that will decay within the next $2^{14}-1$, $2^{20}-1$, and $2^{26}-1$ ticks, respectively.

The tv5 field is identical to the previous ones, except that the last entry of the vec array is a list that includes dynamic timers with extremely large expires fields. It never needs to be replenished from another array. Figure 6-1 illustrates in a schematic way the five groups of lists.

The timer_jiffies field represents the earliest expiration time of the dynamic timers yet to be checked: if it coincides with the value of jiffies, no backlog of deferrable functions has accumulated; if it is smaller than jiffies, then lists of dynamic timers that refer to previous ticks must be dealt with. The field is set to jiffies at system startup and is increased only by the run_timer_softirq() function described in the next section. Notice that the timer_jiffies field might drop a long way behind jiffies when the deferrable functions that handle dynamic timers are not executed
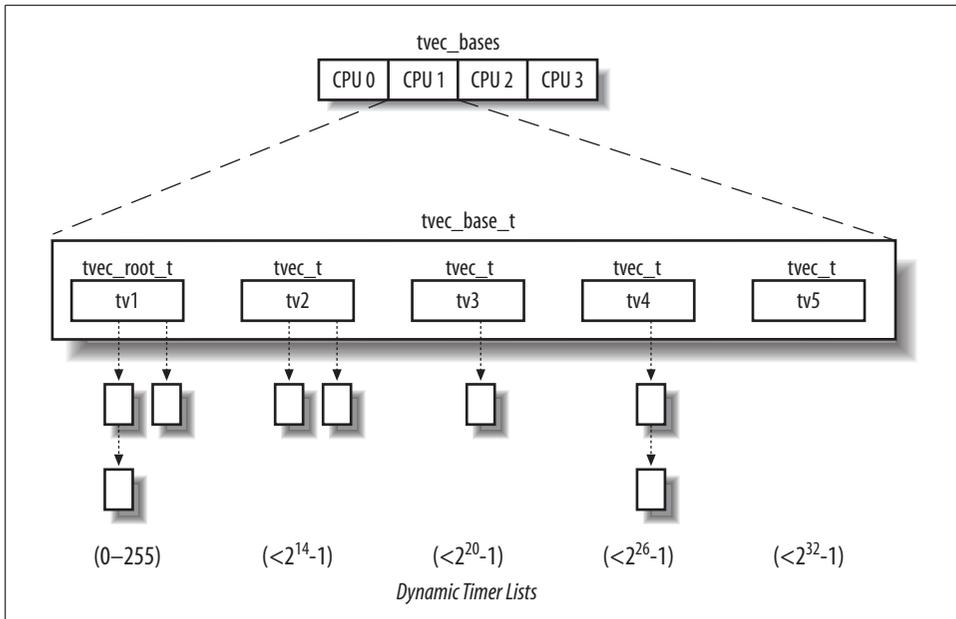
Figure 6-1. The groups of lists associated with dynamic timers

for a long time—for instance because these functions have been disabled or because a large number of interrupt handlers have been executed.

In multiprocessor systems, the running_timer field points to the timer_list structure of the dynamic timer that is currently handled by the local CPU.

Dynamic timer handling

Despite the clever data structures, handling software timers is a time-consuming activity that should not be performed by the timer interrupt handler. In Linux 2.6 this activity is carried on by a deferrable function, namely the TIMER_SOFTIRQ softirq.

The run_timer_softirq() function is the deferrable function associated with the TIMER_SOFTIRQ softirq. It essentially performs the following actions:

1. Stores in the base local variable the address of the tvec_base_t data structure associated with the local CPU.

2. Acquires the base->lock spin lock and disables local interrupts.

3. Starts a while loop, which ends when base->timer_jiffies becomes greater than the value of jiffies. In every single execution of the cycle, performs the following substeps:

   a. Computes the index of the list in base->tv1 that holds the next timers to be handled:

   ```
   index = base->timer_jiffies & 255;
   ```

b. If `index` is zero, all lists in `base->tv1` have been checked, so they are empty: the function therefore percolates the dynamic timers by invoking `cascade()`:

```
if (!index &&
    (!cascade(base, &base->tv2, (base->timer_jiffies>> 8)&63)) &&
    (!cascade(base, &base->tv3, (base->timer_jiffies>>14)&63)) &&
    (!cascade(base, &base->tv4, (base->timer_jiffies>>20)&63)))
      cascade(base, &base->tv5, (base->timer_jiffies>>26)&63);
```

Consider the first invocation of the `cascade()` function: it receives as arguments the address in base, the address of `base->tv2`, and the index of the list in `base->tv2` including the timers that will decay in the next 256 ticks. This index is determined by looking at the proper bits of the `base->timer_jiffies` value. `cascade()` moves all dynamic timers in the `base->tv2` list into the proper lists of `base->tv1`; then, it returns a positive value, unless all `base->tv2` lists are now empty. If so, `cascade()` is invoked once more to replenish `base->tv2` with the timers included in a list of `base->tv3`, and so on.

c. Increases by one `base->timer_jiffies`.

d. For each dynamic timer in the `base->tv1.vec[index]` list, executes the corresponding timer function. In particular, for each `timer_list` element t in the list essentially performs the following steps:

1. Removes t from the `base->tv1`'s list.

2. In multiprocessor systems, sets `base->running_timer` to &t.

3. Sets `t.base` to NULL.

4. Releases the `base->lock` spin lock, and enables local interrupts.

5. Executes the timer function `t.function` passing as argument `t.data`.

6. Acquires the `base->lock` spin lock, and disables local interrupts.

7. Continues with the next timer in the list, if any.

e. All timers in the list have been handled. Continues with the next iteration of the outermost `while` cycle.

4. The outermost `while` cycle is terminated, which means that all decayed timers have been handled. In multiprocessor systems, sets `base->running_timer` to NULL.

5. Releases the `base->lock` spin lock and enables local interrupts.

Because the values of `jiffies` and `timer_jiffies` usually coincide, the outermost `while` cycle is often executed only once. In general, the outermost loop is executed `jiffies - base->timer_jiffies + 1` consecutive times. Moreover, if a timer interrupt occurs while `run_timer_softirq()` is being executed, dynamic timers that decay at this tick occurrence are also considered, because the `jiffies` variable is asynchronously increased by the global timer interrupt handler (see the earlier section "The timer interrupt handler").

Notice that `run_timer_softirq()` disables interrupts and acquires the `base->lock` spin lock just before entering the outermost loop; interrupts are enabled and the spin lock

is released right before invoking each dynamic timer function, until its termination. This ensures that the dynamic timer data structures are not corrupted by interleaved kernel control paths.

To sum up, this rather complex algorithm ensures excellent performance. To see why, assume for the sake of simplicity that the TIMER_SOFTIRQ softirq is executed right after the corresponding timer interrupt occurs. Then, in 255 timer interrupt occurrences out of 256 (in 99.6% of the cases), the run_timer_softirq( ) function just runs the functions of the decayed timers, if any. To replenish base->tv1.vec periodically, it is sufficient 63 times out of 64 to partition one list of base->tv2 into the 256 lists of base->tv1. The base->tv2.vec array, in turn, must be replenished in 0.006 percent of the cases (that is, once every 16.4 seconds). Similarly, base->tv3.vec is replenished every 17 minutes and 28 seconds, and base->tv4.vec is replenished every 18 hours and 38 minutes. base->tv5.vec doesn't need to be replenished.

An Application of Dynamic Timers: the nanosleep( ) System Call

To show how the outcomes of all the previous activities are actually used in the kernel, we'll show an example of the creation and use of a *process time-out*.

Let's consider the service routine of the nanosleep() system call, that is, sys_nanosleep(), which receives as its parameter a pointer to a timespec structure and suspends the invoking process until the specified time interval elapses. The service routine first invokes copy_from_user() to copy the values contained in the User Mode timespec structure into the local variable t. Assuming that the timespec structure defines a non-null delay, the function then executes the following code:

```
current->state = TASK_INTERRUPTIBLE;
remaining = schedule_timeout(timespec_to_jiffies(&t)+1);
```

The timespec_to_jiffies( ) function converts in ticks the time interval stored in the timespec structure. To be on the safe side, sys_nanosleep( ) adds one tick to the value computed by timespec_to_jiffies( ).

The kernel implements process time-outs by using dynamic timers. They appear in the schedule_timeout( ) function, which essentially executes the following statements:

```
struct timer_list timer;
unsigned long expire = timeout + jiffies;
init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;
add_timer(&timer);
schedule();     /* process suspended until timer expires */
del_singleshot_timer_sync(&timer);
timeout = expire - jiffies;
return (timeout < 0 ? 0 : timeout);
```

When `schedule()` is invoked, another process is selected for execution; when the former process resumes its execution, the function removes the dynamic timer. In the last statement, the function either returns 0, if the time-out is expired, or the number of ticks left to the time-out expiration if the process was awakened for some other reason.

When the time-out expires, the timer's function is executed:

```
void process_timeout(unsigned long __data)
{
    wake_up_process((task_t *)__data);
}
```

The `process_timeout()` receives as its parameter the process descriptor pointer stored in the `data` field of the `timer` object. As a result, the suspended process is awakened.

Once awakened, the process continues the execution of the `sys_nanosleep()` system call. If the value returned by `schedule_timeout()` specifies that the process time-out is expired (value zero), the system call terminates. Otherwise, the system call is automatically restarted, as explained in the section "Reexecution of System Calls" in Chapter 11.

Delay Functions

Software timers are useless when the kernel must wait for a short time interval—let's say, less than a few milliseconds. For instance, often a device driver has to wait for a predefined number of microseconds until the hardware completes some operation. Because a dynamic timer has a significant setup overhead and a rather large minimum wait time (1 millisecond), the device driver cannot conveniently use it.

In these cases, the kernel makes use of the `udelay()` and `ndelay()` functions: the former receives as its parameter a time interval in microseconds and returns after the specified delay has elapsed; the latter is similar, but the argument specifies the delay in nanoseconds.

Essentially, the two functions are defined as follows:

```
void udelay(unsigned long usecs)
{
    unsigned long loops;
    loops = (usecs*HZ*current_cpu_data.loops_per_jiffy)/1000000;
    cur_timer->delay(loops);
}

void ndelay(unsigned long nsecs)
{
    unsigned long loops;
    loops = (nsecs*HZ*current_cpu_data.loops_per_jiffy)/1000000000;
    cur_timer->delay(loops);
}
```

Both functions rely on the `delay` method of the `cur_timer` timer object (see the earlier section "Data Structures of the Timekeeping Architecture"), which receives as its parameter a time interval in "loops." The exact duration of one "loop," however, depends on the timer object referred by `cur_timer` (see Table 6-2 earlier in this chapter):

- If `cur_timer` points to the `timer_hpet`, `timer_pmtmr`, and `timer_tsc` objects, one "loop" corresponds to one CPU cycle—that is, the time interval between two consecutive CPU clock signals (see the earlier section "Time Stamp Counter (TSC)").

- If `cur_timer` points to the `timer_none` or `timer_pit` objects, one "loop" corresponds to the time duration of a single iteration of a tight instruction loop.

During the initialization phase, after `cur_timer` has been set up by `select_timer()`, the kernel executes the `calibrate_delay()` function, which determines how many "loops" fit in a tick. This value is then saved in the `current_cpu_data.loops_per_jiffy` variable, so that it can be used by `udelay()` and `ndelay()` to convert microseconds and nanoseconds, respectively, to "loops."

Of course, the `cur_timer->delay()` method makes use of the HPET or TSC hardware circuitry, if available, to get an accurate measurement of time. Otherwise, if no HPET or TSC is available, the method executes `loops` iterations of a tight instruction loop.

System Calls Related to Timing Measurements

Several system calls allow User Mode processes to read and modify the time and date and to create timers. Let's briefly review these and discuss how the kernel handles them.

The time( ) and gettimeofday( ) System Calls

Processes in User Mode can get the current time and date by means of several system calls:

`time( )`
> Returns the number of elapsed seconds since midnight at the start of January 1, 1970 (UTC).

`gettimeofday( )`
> Returns, in a data structure named `timeval`, the number of elapsed seconds since midnight of January 1, 1970 (UTC) and the number of elapsed microseconds in the last second (a second data structure named `timezone` is not currently used).

The `time( )` system call is superseded by `gettimeofday( )`, but it is still included in Linux for backward compatibility. Another widely used function, `ftime( )`, which is no longer implemented as a system call, returns the number of elapsed seconds since

midnight of January 1, 1970 (UTC) and the number of elapsed milliseconds in the last second.

The gettimeofday( ) system call is implemented by the sys_gettimeofday( ) function. To compute the current date and time of the day, this function invokes do_ gettimeofday( ), which executes the following actions:

1. Acquires the xtime_lock seqlock for reading.
2. Determines the number of microseconds elapsed since the last timer interrupt by invoking the get_offset method of the cur_timer timer object:

   ```
   usec = cur_timer->getoffset();
   ```

   As explained in the earlier section "Data Structures of the Timekeeping Architecture," there are four possible cases:

   a. If cur_timer points to the timer_hpet object, the method compares the current value of the HPET counter with the value of the same counter saved in the last execution of the timer interrupt handler.

   b. If cur_timer points to the timer_pmtmr object, the method compares the current value of the ACPI PMT counter with the value of the same counter saved in the last execution of the timer interrupt handler.

   c. If cur_timer points to the timer_tsc object, the method compares the current value of the Time Stamp Counter with the value of the TSC saved in the last execution of the timer interrupt handler.

   d. If cur_timer points to the timer_pit object, the method reads the current value of the PIT counter to compute the number of microseconds elapsed since the last PIT's timer interrupt.

3. If some timer interrupt has been lost (see the section "Updating the Time and Date" earlier in this chapter), the function adds to usec the corresponding delay:

   ```
   usec += (jiffies - wall_jiffies) * 1000;
   ```

4. Adds to usec the microseconds elapsed in the last second:

   ```
   usec += (xtime.tv_nsec / 1000);
   ```

5. Copies the contents of xtime into the user-space buffer specified by the system call parameter tv, adding to the microseconds field the value of usec:

   ```
   tv->tv_sec = xtime->tv_sec;
   tv->tv_usec = usec;
   ```

6. Invokes read_seqretry( ) on the xtime_lock seqlock, and jumps back to step 1 if another kernel control path has concurrently acquired xtime_lock for writing.

7. Checks for an overflow in the microseconds field, adjusting both that field and the second field if necessary:

   ```
   while (tv->tv_usec >= 1000000) {
       tv->tv_usec -= 1000000;
       tv->tv_sec++;
   }
   ```

Processes in User Mode with root privilege may modify the current date and time by using either the obsolete stime( ) or the settimeofday( ) system call. The sys_settimeofday( ) function invokes do_settimeofday( ), which executes operations complementary to those of do_gettimeofday( ).

Notice that both system calls modify the value of xtime while leaving the RTC registers unchanged. Therefore, the new time is lost when the system shuts down, unless the user executes the *clock* program to change the RTC value.

The adjtimex( ) System Call

Although clock drift ensures that all systems eventually move away from the correct time, changing the time abruptly is both an administrative nuisance and risky behavior. Imagine, for instance, programmers trying to build a large program and depending on file timestamps to make sure that out-of-date object files are recompiled. A large change in the system's time could confuse the make program and lead to an incorrect build. Keeping the clocks tuned is also important when implementing a distributed filesystem on a network of computers. In this case, it is wise to adjust the clocks of the interconnected PCs, so that the timestamp values associated with the inodes of the accessed files are coherent. Thus, systems are often configured to run a time synchronization protocol such as Network Time Protocol (NTP) on a regular basis to change the time gradually at each tick. This utility depends on the adjtimex( ) system call in Linux.

This system call is present in several Unix variants, although it should not be used in programs intended to be portable. It receives as its parameter a pointer to a timex structure, updates kernel parameters from the values in the timex fields, and returns the same structure with current kernel values. Such kernel values are used by update_wall_time_one_tick( ) to slightly adjust the number of microseconds added to xtime.tv_usec at each tick.

The setitimer( ) and alarm( ) System Calls

Linux allows User Mode processes to activate special timers called *interval timers*.[*] The timers cause Unix signals (see Chapter 11) to be sent periodically to the process. It is also possible to activate an interval timer so that it sends just one signal after a specified delay. Each interval timer is therefore characterized by:

- The frequency at which the signals must be emitted, or a null value if just one signal has to be generated
- The time remaining until the next signal is to be generated

---

[*] These software constructs have nothing in common with the Programmable Interval Timer chip described earlier in this chapter.

The earlier warning about accuracy applies to these timers. They are guaranteed to execute after the requested time has elapsed, but it is impossible to predict exactly when they will be delivered.

Interval timers are activated by means of the POSIX `setitimer()` system call. The first parameter specifies which of the following policies should be adopted:

`ITIMER_REAL`
> The actual elapsed time; the process receives `SIGALRM` signals.

`ITIMER_VIRTUAL`
> The time spent by the process in User Mode; the process receives `SIGVTALRM` signals.

`ITIMER_PROF`
> The time spent by the process both in User and in Kernel Mode; the process receives `SIGPROF` signals.

The interval timers can be either single-shot or periodic. The second parameter of `setitimer()` points to a structure of type `itimerval` that specifies the initial duration of the timer (in seconds and nanoseconds) and the duration to be used when the timer is automatically reactivated (or zero for single-shot timers). The third parameter of `setitimer()` is an optional pointer to an `itimerval` structure that is filled by the system call with the previous timer parameters.

To implement an interval timer for each of the preceding policies, the process descriptor includes three pairs of fields:

- `it_real_incr` and `it_real_value`
- `it_virt_incr` and `it_virt_value`
- `it_prof_incr` and `it_prof_value`

The first field of each pair stores the interval in ticks between two signals; the other field stores the current value of the timer.

The `ITIMER_REAL` interval timer is implemented by using dynamic timers because the kernel must send signals to the process even when it is not running on the CPU. Therefore, each process descriptor includes a dynamic timer object called `real_timer`. The `setitimer()` system call initializes the `real_timer` fields and then invokes `add_timer()` to insert the dynamic timer in the proper list. When the timer expires, the kernel executes the `it_real_fn()` timer function. In turn, the `it_real_fn()` function sends a `SIGALRM` signal to the process; then, if `it_real_incr` is not null, it sets the `expires` field again, reactivating the timer.

The `ITIMER_VIRTUAL` and `ITIMER_PROF` interval timers do not require dynamic timers, because they can be updated while the process is running. The `account_it_virt()` and `account_it_prof()` functions are invoked by `update_process_times()`, which is called either by the PIT's timer interrupt handler (UP) or by the local timer interrupt

handlers (SMP). Therefore, the two interval timers are updated once every tick, and if they are expired, the proper signal is sent to the current process.

The `alarm()` system call sends a `SIGALRM` signal to the calling process when a specified time interval has elapsed. It is very similar to `setitimer()` when invoked with the `ITIMER_REAL` parameter, because it uses the `real_timer` dynamic timer included in the process descriptor. Therefore, `alarm()` and `setitimer()` with parameter `ITIMER_REAL` cannot be used at the same time.

System Calls for POSIX Timers

The POSIX 1003.1b standard introduced a new type of software timers for User Mode programs—in particular, for multithreaded and real-time applications. These timers are often referred to as *POSIX timers*.

Every implementation of the POSIX timers must offer to the User Mode programs a few *POSIX clocks*, that is, virtual time sources having predefined resolutions and properties. Whenever an application wants to make use of a POSIX timer, it creates a new timer resource specifying one of the existing POSIX clocks as the timing base. The system calls that allow users to handle POSIX clocks and timers are listed in Table 6-3.

Table 6-3. System calls for POSIX timers and clocks

System call	Description
`clock_gettime()`	Gets the current value of a POSIX clock
`clock_settime()`	Sets the current value of a POSIX clock
`clock_getres()`	Gets the resolution of a POSIX clock
`timer_create()`	Creates a new POSIX timer based on a specified POSIX clock
`timer_gettime()`	Gets the current value and increment of a POSIX timer
`timer_settime()`	Sets the current value and increment of a POSIX timer
`timer_getoverrun()`	Gets the number of overruns of a decayed POSIX timer
`timer_delete()`	Destroys a POSIX timer
`clock_nanosleep()`	Puts the process to sleep using a POSIX clock as time source

The Linux 2.6 kernel offers two types of POSIX clocks:

CLOCK_REALTIME
> This virtual clock represents the real-time clock of the system—essentially the value of the `xtime` variable (see the earlier section "Updating the Time and Date"). The resolution returned by the `clock_getres()` system call is 999,848 nanoseconds, which corresponds to roughly 1000 updates of `xtime` in a second.

CLOCK_MONOTONIC
> This virtual clock represents the real-time clock of the system purged of every time warp due to the synchronization with an external time source. Essentially,

this virtual clock is represented by the sum of the two variables `xtime` and `wall_to_monotonic` (see the earlier section "Timekeeping Architecture in Uniprocessor Systems"). The resolution of this POSIX clock, returned by `clock_getres()`, is 999,848 nanoseconds.

The Linux kernel implements the POSIX timers by means of dynamic timers. Thus, they are similar to the traditional `ITIMER_REAL` interval timers we described in the previous section. POSIX timers, however, are much more flexible and reliable than traditional interval timers. A couple of significant differences between them are:

- When a traditional interval timer decays, the kernel always sends a `SIGALRM` signal to the process that activated the timer. Instead, when a POSIX timer decays, the kernel can send every kind of signal, either to the whole multithreaded application or to a single specified thread. The kernel can also force the execution of a notifier function in a thread of the application, or it can even do nothing (it is up to a User Mode library to handle the event).

- If a traditional interval timer decays many times but the User Mode process cannot receive the `SIGALRM` signal (for instance because the signal is blocked or the process is not running), only the first signal is received: all other occurrences of `SIGALRM` are lost. The same holds for POSIX timers, but the process can invoke the `timer_getoverrun()` system call to get the number of times the timer decayed since the generation of the first signal.

CHAPTER 7

Process Scheduling

Like every time sharing system, Linux achieves the magical effect of an apparent simultaneous execution of multiple processes by switching from one process to another in a very short time frame. Process switching itself was discussed in Chapter 3; this chapter deals with *scheduling*, which is concerned with when to switch and which process to choose.

The chapter consists of three parts. The section "Scheduling Policy" introduces the choices made by Linux in the abstract to schedule processes. The section "The Scheduling Algorithm" discusses the data structures used to implement scheduling and the corresponding algorithm. Finally, the section "System Calls Related to Scheduling" describes the system calls that affect process scheduling.

To simplify the description, we refer as usual to the 80×86 architecture; in particular, we assume that the system uses the Uniform Memory Access model, and that the system tick is set to 1 ms.

Scheduling Policy

The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and so on. The set of rules used to determine when and how to select a new process to run is called *scheduling policy*.

Linux scheduling is based on the *time sharing* technique: several processes run in "time multiplexing" because the CPU time is divided into *slices*, one for each runnable process.[*] Of course, a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or *quantum* expires, a

---

[*] Recall that stopped and suspended processes cannot be selected by the scheduling algorithm to run on a CPU.

process switch may take place. Time sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs to ensure CPU time sharing.

The scheduling policy is also based on ranking processes according to their priority. Complicated algorithms are sometimes used to derive the current priority of a process, but the end result is the same: each process is associated with a value that tells the scheduler how appropriate it is to let the process run on a CPU.

In Linux, process priority is dynamic. The scheduler keeps track of what processes are doing and adjusts their priorities periodically; in this way, processes that have been denied the use of a CPU for a long time interval are boosted by dynamically increasing their priority. Correspondingly, processes running for a long time are penalized by decreasing their priority.

When speaking about scheduling, processes are traditionally classified as *I/O-bound* or *CPU-bound*. The former make heavy use of I/O devices and spend much time waiting for I/O operations to complete; the latter carry on number-crunching applications that require a lot of CPU time.

An alternative classification distinguishes three classes of processes:

Interactive processes
> These interact constantly with their users, and therefore spend a lot of time waiting for keypresses and mouse operations. When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive. Typically, the average delay must fall between 50 and 150 milliseconds. The variance of such delay must also be bounded, or the user will find the system to be erratic. Typical interactive programs are command shells, text editors, and graphical applications.

Batch processes
> These do not need user interaction, and hence they often run in the background. Because such processes do not need to be very responsive, they are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines, and scientific computations.

Real-time processes
> These have very stringent scheduling requirements. Such processes should never be blocked by lower-priority processes and should have a short guaranteed response time with a minimum variance. Typical real-time programs are video and sound applications, robot controllers, and programs that collect data from physical sensors.

The two classifications we just offered are somewhat independent. For instance, a batch process can be either I/O-bound (e.g., a database server) or CPU-bound (e.g., an image-rendering program). While real-time programs are explicitly recognized as such by the scheduling algorithm in Linux, there is no easy way to distinguish

between interactive and batch programs. The Linux 2.6 scheduler implements a sophisticated heuristic algorithm based on the past behavior of the processes to decide whether a given process should be considered as interactive or batch. Of course, the scheduler tends to favor interactive processes over batch ones.

Programmers may change the scheduling priorities by means of the system calls illustrated in Table 7-1. More details are given in the section "System Calls Related to Scheduling."

Table 7-1. System calls related to scheduling

System call	Description
nice( )	Change the static priority of a conventional process
getpriority( )	Get the maximum static priority of a group of conventional processes
setpriority( )	Set the static priority of a group of conventional processes
sched_getscheduler( )	Get the scheduling policy of a process
sched_setscheduler( )	Set the scheduling policy and the real-time priority of a process
sched_getparam( )	Get the real-time priority of a process
sched_setparam( )	Set the real-time priority of a process
sched_yield( )	Relinquish the processor voluntarily without blocking
sched_get_priority_min( )	Get the minimum real-time priority value for a policy
sched_get_priority_max( )	Get the maximum real-time priority value for a policy
sched_rr_get_interval( )	Get the time quantum value for the Round Robin policy
sched_setaffinity( )	Set the CPU affinity mask of a process
sched_getaffinity( )	Get the CPU affinity mask of a process

Process Preemption

As mentioned in the first chapter, Linux processes are *preemptable*. When a process enters the TASK_RUNNING state, the kernel checks whether its dynamic priority is greater than the priority of the currently running process. If it is, the execution of current is interrupted and the scheduler is invoked to select another process to run (usually the process that just became runnable). Of course, a process also may be preempted when its time quantum expires. When this occurs, the TIF_NEED_RESCHED flag in the thread_info structure of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.

For instance, let's consider a scenario in which only two programs—a text editor and a compiler—are being executed. The text editor is an interactive program, so it has a higher dynamic priority than the compiler. Nevertheless, it is often suspended, because the user alternates between pauses for think time and data entry; moreover, the average delay between two keypresses is relatively long. However, as soon as the user presses a key, an interrupt is raised and the kernel wakes up the text editor process. The kernel also determines that the dynamic priority of the editor is higher than

the priority of current, the currently running process (the compiler), so it sets the TIF_NEED_RESCHED flag of this process, thus forcing the scheduler to be activated when the kernel finishes handling the interrupt. The scheduler selects the editor and performs a process switch; as a result, the execution of the editor is resumed very quickly and the character typed by the user is echoed to the screen. When the character has been processed, the text editor process suspends itself waiting for another keypress and the compiler process can resume its execution.

Be aware that a preempted process is not suspended, because it remains in the TASK_RUNNING state; it simply no longer uses the CPU. Moreover, remember that the Linux 2.6 kernel is preemptive, which means that a process can be preempted either when executing in Kernel or in User Mode; we discussed in depth this feature in the section "Kernel Preemption" in Chapter 5.

How Long Must a Quantum Last?

The quantum duration is critical for system performance: it should be neither too long nor too short.

If the average quantum duration is too short, the system overhead caused by process switches becomes excessively high. For instance, suppose that a process switch requires 5 milliseconds; if the quantum is also set to 5 milliseconds, then at least 50 percent of the CPU cycles will be dedicated to process switching.*

If the average quantum duration is too long, processes no longer appear to be executed concurrently. For instance, let's suppose that the quantum is set to five seconds; each runnable process makes progress for about five seconds, but then it stops for a very long time (typically, five seconds times the number of runnable processes).

It is often believed that a long quantum duration degrades the response time of interactive applications. This is usually false. As described in the section "Process Preemption" earlier in this chapter, interactive processes have a relatively high priority, so they quickly preempt the batch processes, no matter how long the quantum duration is.

In some cases, however, a very long quantum duration degrades the responsiveness of the system. For instance, suppose two users concurrently enter two commands at the respective shell prompts; one command starts a CPU-bound process, while the other launches an interactive application. Both shells fork a new process and delegate the execution of the user's command to it; moreover, suppose such new processes have the same initial priority (Linux does not know in advance if a program to be executed is batch or interactive). Now if the scheduler selects the CPU-bound process to run

---

* Actually, things could be much worse than this; for example, if the time required for the process switch is counted in the process quantum, all CPU time is devoted to the process switch and no process can progress toward its termination.

first, the other process could wait for a whole time quantum before starting its execution. Therefore, if the quantum duration is long, the system could appear to be unresponsive to the user that launched the interactive application.

The choice of the average quantum duration is always a compromise. The rule of thumb adopted by Linux is choose a duration as long as possible, while keeping good system response time.

The Scheduling Algorithm

The scheduling algorithm used in earlier versions of Linux was quite simple and straightforward: at every process switch the kernel scanned the list of runnable processes, computed their priorities, and selected the "best" process to run. The main drawback of that algorithm is that the time spent in choosing the best process depends on the number of runnable processes; therefore, the algorithm is too costly—that is, it spends too much time—in high-end systems running thousands of processes.

The scheduling algorithm of Linux 2.6 is much more sophisticated. By design, it scales well with the number of runnable processes, because it selects the process to run in constant time, independently of the number of runnable processes. It also scales well with the number of processors because each CPU has its own queue of runnable processes. Furthermore, the new algorithm does a better job of distinguishing interactive processes and batch processes. As a consequence, users of heavily loaded systems feel that interactive applications are much more responsive in Linux 2.6 than in earlier versions.

The scheduler always succeeds in finding a process to be executed; in fact, there is always at least one runnable process: the *swapper* process, which has PID 0 and executes only when the CPU cannot execute other processes. As mentioned in Chapter 3, every CPU of a multiprocessor system has its own *swapper* process with PID equal to 0.

Every Linux process is always scheduled according to one of the following *scheduling classes*:

SCHED_FIFO
> A First-In, First-Out real-time process. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list. If no other higher-priority real-time process is runnable, the process continues to use the CPU as long as it wishes, even if other real-time processes that have the same priority are runnable.

SCHED_RR
> A Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list. This policy ensures a fair assignment of CPU time to all SCHED_RR real-time processes that have the same priority.

SCHED_NORMAL

A conventional, time-shared process.

The scheduling algorithm behaves quite differently depending on whether the process is conventional or real-time.

Scheduling of Conventional Processes

Every conventional process has its own *static priority*, which is a value used by the scheduler to rate the process with respect to the other conventional processes in the system. The kernel represents the static priority of a conventional process with a number ranging from 100 (highest priority) to 139 (lowest priority); notice that static priority decreases as the values increase.

A new process always inherits the static priority of its parent. However, a user can change the static priority of the processes that he owns by passing some "nice values" to the nice( ) and setpriority( ) system calls (see the section "System Calls Related to Scheduling" later in this chapter).

Base time quantum

The static priority essentially determines the *base time quantum* of a process, that is, the time quantum duration assigned to the process when it has exhausted its previous time quantum. Static priority and base time quantum are related by the following formula:

$$\begin{matrix} base\ time\ quantum \\ (in\ milliseconds) \end{matrix} = \begin{cases} (140 - static\ priority) \times 20 & if\ static\ priority < 120 \\ (140 - static\ priority) \times\ 5 & if\ static\ priority \geq 120 \end{cases} \quad (1)$$

As you see, the higher the static priority (i.e., the lower its numerical value), the longer the base time quantum. As a consequence, higher priority processes usually get longer slices of CPU time with respect to lower priority processes. Table 7-2 shows the static priority, the base time quantum values, and the corresponding nice values for a conventional process having highest static priority, default static priority, and lowest static priority. (The table also lists the values of the interactive delta and of the sleep time threshold, which are explained later in this chapter.)

Table 7-2. Typical priority values for a conventional process

Description	Static priority	Nice value	Base time quantum	Interactivedelta	Sleep time threshold
Highest static priority	100	−20	800 ms	−3	299 ms
High static priority	110	- 10	600 ms	- 1	499 ms
Default static priority	120	0	100 ms	+2	799 ms
Low static priority	130	+10	50 ms	+4	999 ms
Lowest static priority	139	+19	5 ms	+6	1199 ms

Dynamic priority and average sleep time

Besides a static priority, a conventional process also has a *dynamic priority*, which is a value ranging from 100 (highest priority) to 139 (lowest priority). The dynamic priority is the number actually looked up by the scheduler when selecting the new process to run. It is related to the static priority by the following empirical formula:

$$dynamic\ priority = \max(100, \min(static\ priority - bonus + 5, 139)) \qquad (2)$$

The *bonus* is a value ranging from 0 to 10; a value less than 5 represents a penalty that lowers the dynamic priority, while a value greater than 5 is a premium that raises the dynamic priority. The value of the bonus, in turn, depends on the past history of the process; more precisely, it is related to the *average sleep time* of the process.

Roughly, the average sleep time is the average number of nanoseconds that the process spent while sleeping. Be warned, however, that this is not an average operation on the elapsed time. For instance, sleeping in TASK_INTERRUPTIBLE state contributes to the average sleep time in a different way from sleeping in TASK_UNINTERRUPTIBLE state. Moreover, the average sleep time decreases while a process is running. Finally, the average sleep time can never become larger than 1 second.

The correspondence between average sleep times and bonus values is shown in Table 7-3. (The table lists also the corresponding granularity of the time slice, which will be discussed later.)

Table 7-3. Average sleep times, bonus values, and time slice granularity

Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10

The average sleep time is also used by the scheduler to determine whether a given process should be considered interactive or batch. More precisely, a process is considered "interactive" if it satisfies the following formula:

$$dynamic\ priority \le 3 \times static\ priority\ /\ 4 + 28 \qquad (3)$$

which is equivalent to the following:

> *bonus* - 5 ≥ *static priority* / 4 − 28

The expression *static priority* / 4 − 28 is called the *interactive delta*; some typical values of this term are listed in Table 7-2. It should be noted that it is far easier for high priority than for low priority processes to become interactive. For instance, a process having highest static priority (100) is considered interactive when its bonus value exceeds 2, that is, when its average sleep time exceeds 200 ms. Conversely, a process having lowest static priority (139) is never considered as interactive, because the bonus value is always smaller than the value 11 required to reach an interactive delta equal to 6. A process having default static priority (120) becomes interactive as soon as its average sleep time exceeds 700 ms.

Active and expired processes

Even if conventional processes having higher static priorities get larger slices of the CPU time, they should not completely lock out the processes having lower static priority. To avoid process starvation, when a process finishes its time quantum, it can be replaced by a lower priority process whose time quantum has not yet been exhausted. To implement this mechanism, the scheduler keeps two disjoint sets of runnable processes:

Active processes
> These runnable processes have not yet exhausted their time quantum and are thus allowed to run.

Expired processes
> These runnable processes have exhausted their time quantum and are thus forbidden to run until all active processes expire.

However, the general schema is slightly more complicated than this, because the scheduler tries to boost the performance of interactive processes. An active batch process that finishes its time quantum always becomes expired. An active interactive process that finishes its time quantum usually remains active: the scheduler refills its time quantum and leaves it in the set of active processes. However, the scheduler moves an interactive process that finished its time quantum into the set of expired processes if the eldest expired process has already waited for a long time, or if an expired process has higher static priority (lower value) than the interactive process. As a consequence, the set of active processes will eventually become empty and the expired processes will have a chance to run.

Scheduling of Real-Time Processes

Every real-time process is associated with a *real-time priority*, which is a value ranging from 1 (highest priority) to 99 (lowest priority). The scheduler always favors a higher priority runnable process over a lower priority one; in other words, a real-time

process inhibits the execution of every lower-priority process while it remains runnable. Contrary to conventional processes, real-time processes are always considered active (see the previous section). The user can change the real-time priority of a process by means of the `sched_setparam()` and `sched_setscheduler()` system calls (see the section "System Calls Related to Scheduling" later in this chapter).

If several real-time runnable processes have the same highest priority, the scheduler chooses the process that occurs first in the corresponding list of the local CPU's runqueue (see the section "The lists of TASK_RUNNING processes" in Chapter 3).

A real-time process is replaced by another process only when one of the following events occurs:

- The process is preempted by another process having higher real-time priority.
- The process performs a blocking operation, and it is put to sleep (in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`).
- The process is stopped (in state `TASK_STOPPED` or `TASK_TRACED`), or it is killed (in state `EXIT_ZOMBIE` or `EXIT_DEAD`).
- The process voluntarily relinquishes the CPU by invoking the `sched_yield()` system call (see the section "System Calls Related to Scheduling" later in this chapter).
- The process is Round Robin real-time (`SCHED_RR`), and it has exhausted its time quantum.

The `nice()` and `setpriority()` system calls, when applied to a Round Robin real-time process, do not change the real-time priority but rather the duration of the base time quantum. In fact, the duration of the base time quantum of Round Robin real-time processes does not depend on the real-time priority, but rather on the static priority of the process, according to the formula (1) in the earlier section "Scheduling of Conventional Processes."

Data Structures Used by the Scheduler

Recall from the section "Identifying a Process" in Chapter 3 that the process list links all process descriptors, while the runqueue lists link the process descriptors of all runnable processes—that is, of those in a `TASK_RUNNING` state—except the *swapper* process (idle process).

The runqueue Data Structure

The `runqueue` data structure is the most important data structure of the Linux 2.6 scheduler. Each CPU in the system has its own runqueue; all `runqueue` structures are stored in the `runqueues` per-CPU variable (see the section "Per-CPU Variables" in Chapter 5). The `this_rq()` macro yields the address of the runqueue of the local CPU, while the `cpu_rq(n)` macro yields the address of the runqueue of the CPU having index `n`.

Table 7-4 lists the fields included in the runqueue data structure; we will discuss most of them in the following sections of the chapter.

Table 7-4. The fields of the runqueue structure

Type	Name	Description
spinlock_t	lock	Spin lock protecting the lists of processes
unsigned long	nr_running	Number of runnable processes in the runqueue lists
unsigned long	cpu_load	CPU load factor based on the average number of processes in the runqueue
unsigned long	nr_switches	Number of process switches performed by the CPU
unsigned long	nr_uninterruptible	Number of processes that were previously in the runqueue lists and are now sleeping in TASK_ UNINTERRUPTIBLE state (only the sum of these fields across all runqueues is meaningful)
unsigned long	expired_timestamp	Insertion time of the eldest process in the expired lists
unsigned long long	timestamp_last_tick	Timestamp value of the last timer interrupt
task_t *	curr	Process descriptor pointer of the currently running process (same as current for the local CPU)
task_t *	idle	Process descriptor pointer of the *swapper* process for this CPU
struct mm_struct *	prev_mm	Used during a process switch to store the address of the memory descriptor of the process being replaced
prio_array_t *	active	Pointer to the lists of active processes
prio_array_t *	expired	Pointer to the lists of expired processes
prio_array_t [2]	arrays	The two sets of active and expired processes
int	best_expired_prio	The best static priority (lowest value) among the expired processes
atomic_t	nr_iowait	Number of processes that were previously in the runqueue lists and are now waiting for a disk I/O operation to complete
struct sched_domain *	sd	Points to the base scheduling domain of this CPU (see the section "Scheduling Domains" later in this chapter)
int	active_balance	Flag set if some process shall be *migrated* from this runqueue to another (runqueue balancing)
int	push_cpu	Not used
task_t *	migration_thread	Process descriptor pointer of the *migration* kernel thread
struct list_head	migration_queue	List of processes to be removed from the runqueue

The most important fields of the runqueue data structure are those related to the lists of runnable processes. Every runnable process in the system belongs to one, and just one, runqueue. As long as a runnable process remains in the same runqueue, it can

be executed only by the CPU owning that runqueue. However, as we'll see later, runnable processes may migrate from one runqueue to another.

The `arrays` field of the runqueue is an array consisting of two `prio_array_t` structures. Each data structure represents a set of runnable processes, and includes 140 doubly linked list heads (one list for each possible process priority), a priority bitmap, and a counter of the processes included in the set (see Table 3-2 in the section Chapter 3).
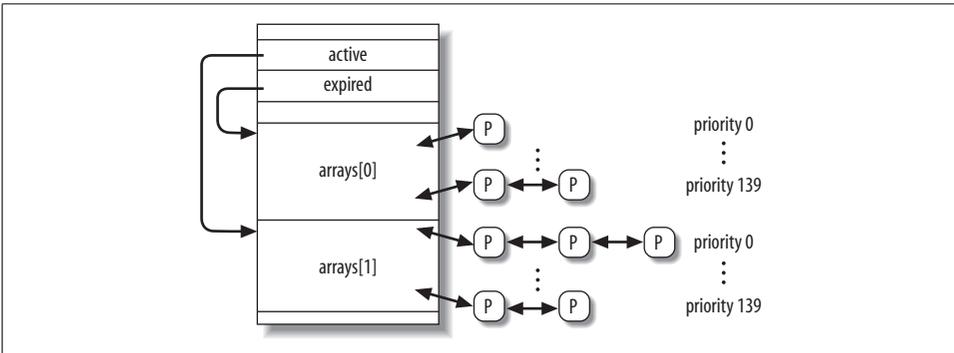
Figure 7-1. The runqueue structure and the two sets of runnable processes

As shown in Figure 7-1, the `active` field of the `runqueue` structure points to one of the two `prio_array_t` data structures in `arrays`: the corresponding set of runnable processes includes the active processes. Conversely, the `expired` field points to the other `prio_array_t` data structure in `arrays`: the corresponding set of runnable processes includes the expired processes.

Periodically, the role of the two data structures in `arrays` changes: the active processes suddenly become the expired processes, and the expired processes become the active ones. To achieve this change, the scheduler simply exchanges the contents of the `active` and `expired` fields of the runqueue.

Process Descriptor

Each process descriptor includes several fields related to scheduling; they are listed in Table 7-5.

Table 7-5. Fields of the process descriptor related to the scheduler

Type	Name	Description
unsigned long	thread_info->flags	Stores the TIF_NEED_RESCHED flag, which is set if the scheduler must be invoked (see the section "Returning from Interrupts and Exceptions" in Chapter 4)
unsigned int	thread_info->cpu	Logical number of the CPU owning the runqueue to which the runnable process belongs

Type	Name	Description
`unsigned long`	`state`	The current state of the process (see the section "Process State" in Chapter 3)
`int`	`prio`	Dynamic priority of the process
`int`	`static_prio`	Static priority of the process
`struct list_head`	`run_list`	Pointers to the next and previous elements in the runqueue list to which the process belongs
`prio_array_t *`	`array`	Pointer to the runqueue's `prio_array_t` set that includes the process
`unsigned long`	`sleep_avg`	Average sleep time of the process
`unsigned long long`	`timestamp`	Time of last insertion of the process in the runqueue, or time of last process switch involving the process
`unsigned long long`	`last_ran`	Time of last process switch that replaced the process
`int`	`activated`	Condition code used when the process is awakened
`unsigned long`	`policy`	The scheduling class of the process (SCHED_NORMAL, SCHED_RR, or SCHED_FIFO)
`cpumask_t`	`cpus_allowed`	Bit mask of the CPUs that can execute the process
`unsigned int`	`time_slice`	Ticks left in the time quantum of the process
`unsigned int`	`first_time_slice`	Flag set to 1 if the process never exhausted its time quantum
`unsigned long`	`rt_priority`	Real-time priority of the process

When a new process is created, `sched_fork()`, invoked by `copy_process()`, sets the `time_slice` field of both `current` (the parent) and `p` (the child) processes in the following way:

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```

In other words, the number of ticks left to the parent is split in two halves: one for the parent and one for the child. This is done to prevent users from getting an unlimited amount of CPU time by using the following method: the parent process creates a child process that runs the same code and then kills itself; by properly adjusting the creation rate, the child process would always get a fresh quantum before the quantum of its parent expires. This programming trick does not work because the kernel does not reward forks. Similarly, a user cannot hog an unfair share of the processor by starting several background processes in a shell or by opening a lot of windows on a graphical desktop. More generally speaking, a process cannot hog resources (unless it has privileges to give itself a real-time policy) by forking multiple descendents.

If the parent had just one tick left in its time slice, the splitting operation forces `current->time_slice` to 0, thus exhausting the quantum of the parent. In this case, `copy_process()` sets `current->time_slice` back to 1, then invokes `scheduler_tick()` to decrease the field (see the following section).

The `copy_process()` function also initializes a few other fields of the child's process descriptor related to scheduling:

```
p->first_time_slice = 1;
p->timestamp = sched_clock();
```

The `first_time_slice` flag is set to 1, because the child has never exhausted its time quantum (if a process terminates or executes a new program during its first time slice, the parent process is rewarded with the remaining time slice of the child). The `timestamp` field is initialized with a timestamp value produced by `sched_clock()`: essentially, this function returns the contents of the 64-bit TSC register (see the section "Time Stamp Counter (TSC)" in Chapter 6) converted to nanoseconds.

Functions Used by the Scheduler

The scheduler relies on several functions in order to do its work; the most important are:

`scheduler_tick()`
    Keeps the `time_slice` counter of `current` up-to-date

`try_to_wake_up()`
    Awakens a sleeping process

`recalc_task_prio()`
    Updates the dynamic priority of a process

`schedule()`
    Selects a new process to be executed

`load_balance()`
    Keeps the runqueues of a multiprocessor system balanced

The scheduler_tick( ) Function

We have already explained in the section "Updating Local CPU Statistics" in Chapter 6 how `scheduler_tick()` is invoked once every tick to perform some operations related to scheduling. It executes the following main steps:

1. Stores in the `timestamp_last_tick` field of the local runqueue the current value of the TSC converted to nanoseconds; this timestamp is obtained from the `sched_clock()` function (see the previous section).

2. Checks whether the current process is the *swapper* process of the local CPU. If so, it performs the following substeps:

    a. If the local runqueue includes another runnable process besides *swapper*, it sets the `TIF_NEED_RESCHED` flag of the current process to force rescheduling. As we'll see in the section "The schedule( ) Function" later in this chapter, if the kernel supports the hyper-threading technology (see the section "Runqueue Balancing in Multiprocessor Systems" later in this chapter), a logical

CPU might be idle even if there are runnable processes in its runqueue, as long as those processes have significantly lower priorities than the priority of a process already executing on another logical CPU associated with the same physical CPU.

b. Jumps to step 7 (there is no need to update the time slice counter of the *swapper* process).

3. Checks whether `current->array` points to the active list of the local runqueue. If not, the process has expired its time quantum, but it has not yet been replaced: sets the `TIF_NEED_RESCHED` flag to force rescheduling, and jumps to step 7.

4. Acquires the `this_rq()->lock` spin lock.

5. Decreases the time slice counter of the current process, and checks whether the quantum is exhausted. The operations performed by the function are quite different according to the scheduling class of the process; we will discuss them in a moment.

6. Releases the `this_rq()->lock` spin lock.

7. Invokes the `rebalance_tick()` function, which should ensure that the runqueues of the various CPUs contain approximately the same number of runnable processes. We will discuss runqueue balancing in the later section "Runqueue Balancing in Multiprocessor Systems."

Updating the time slice of a real-time process

If the current process is a FIFO real-time process, `scheduler_tick()` has nothing to do. In this case, in fact, `current` cannot be preempted by lower or equal priority processes, thus it does not make sense to keep its time slice counter up-to-date.

If `current` is a Round Robin real-time process, `scheduler_tick()` decreases its time slice counter and checks whether the quantum is exhausted:

```
if (current->policy == SCHED_RR && !--current->time_slice) {
    current->time_slice = task_timeslice(current);
    current->first_time_slice = 0;
    set_tsk_need_resched(current);
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
            this_rq()->active->queue+current->prio);
}
```

If the function determines that the time quantum is effectively exhausted, it performs a few operations aimed to ensure that `current` will be preempted, if necessary, as soon as possible.

The first operation consists of refilling the time slice counter of the process by invoking `task_timeslice()`. This function considers the static priority of the process and returns the corresponding base time quantum, according to the formula (1) shown in the earlier section "Scheduling of Conventional Processes." Moreover, the `first_time_slice` field of `current` is cleared: this flag is set by `copy_process()` in the service

routine of the `fork()` system call, and should be cleared as soon as the first time quantum of the process elapses.

Next, `scheduler_tick()` invokes the `set_tsk_need_resched()` function to set the `TIF_NEED_RESCHED` flag of the process. As described in the section "Returning from Interrupts and Exceptions" in Chapter 4, this flag forces the invocation of the `schedule()` function, so that `current` can be replaced by another real-time process having equal (or higher) priority, if any.

The last operation of `scheduler_tick()` consists of moving the process descriptor to the last position of the runqueue active list corresponding to the priority of `current`. Placing `current` in the last position ensures that it will not be selected again for execution until every real-time runnable process having the same priority as `current` will get a slice of the CPU time. This is the meaning of round-robin scheduling. The descriptor is moved by first invoking `list_del()` to remove the process from the runqueue active list, then by invoking `list_add_tail()` to insert back the process in the last position of the same list.

Updating the time slice of a conventional process

If the current process is a conventional process, the `scheduler_tick()` function performs the following operations:

1. Decreases the time slice counter (`current->time_slice`).

2. Checks the time slice counter. If the time quantum is exhausted, the function performs the following operations:

   a. Invokes `dequeue_task()` to remove `current` from the `this_rq()->active` set of runnable processes.

   b. Invokes `set_tsk_need_resched()` to set the `TIF_NEED_RESCHED` flag.

   c. Updates the dynamic priority of `current`:

      ```
      current->prio = effective_prio(current);
      ```

      The `effective_prio()` function reads the `static_prio` and `sleep_avg` fields of `current`, and computes the dynamic priority of the process according to the formula (2) shown in the earlier section "Scheduling of Conventional Processes."

   d. Refills the time quantum of the process:

      ```
      current->time_slice = task_timeslice(current);
      current->first_time_slice = 0;
      ```

   e. If the `expired_timestamp` field of the local runqueue data structure is equal to zero (that is, the set of expired processes is empty), writes into the field the value of the current tick:

      ```
      if (!this_rq()->expired_timestamp)
          this_rq()->expired_timestamp = jiffies;
      ```

   f. Inserts the current process either in the active set or in the expired set:

```
    if (!TASK_INTERACTIVE(current) || EXPIRED_STARVING(this_rq())) {
        enqueue_task(current, this_rq()->expired);
        if (current->static_prio < this_rq()->best_expired_prio)
            this_rq()->best_expired_prio = current->static_prio;
    } else
        enqueue_task(current, this_rq()->active);
```

The `TASK_INTERACTIVE` macro yields the value one if the process is recognized as interactive using the formula (3) shown in the earlier section "Scheduling of Conventional Processes." The `EXPIRED_STARVING` macro checks whether the first expired process in the runqueue had to wait for more than 1000 ticks times the number of runnable processes in the runqueue plus one; if so, the macro yields the value one. The `EXPIRED_STARVING` macro also yields the value one if the static priority value of the current process is greater than the static priority value of an already expired process.

3. Otherwise, if the time quantum is not exhausted (`current->time_slice` is not zero), checks whether the remaining time slice of the current process is too long:

```
    if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
            p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
            (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
            (p->array == rq->active)) {
        list_del(&current->run_list);
        list_add_tail(&current->run_list,
                    this_rq()->active->queue+current->prio);
        set_tsk_need_resched(p);
    }
```

The `TIMESLICE_GRANULARITY` macro yields the product of the number of CPUs in the system and a constant proportional to the bonus of the current process (see Table 7-3 earlier in the chapter). Basically, the time quantum of interactive processes with high static priorities is split into several pieces of `TIMESLICE_GRANULARITY` size, so that they do not monopolize the CPU.

The try_to_wake_up( ) Function

The `try_to_wake_up( )` function awakes a sleeping or stopped process by setting its state to `TASK_RUNNING` and inserting it into the runqueue of the local CPU. For instance, the function is invoked to wake up processes included in a wait queue (see the section "How Processes Are Organized" in Chapter 3) or to resume execution of processes waiting for a signal (see Chapter 11). The function receives as its parameters:

- The descriptor pointer (`p`) of the process to be awakened
- A mask of the process states (`state`) that can be awakened
- A flag (`sync`) that forbids the awakened process to preempt the process currently running on the local CPU

The function performs the following operations:

1. Invokes the `task_rq_lock()` function to disable local interrupts and to acquire the lock of the runqueue `rq` owned by the CPU that was last executing the process (it could be different from the local CPU). The logical number of that CPU is stored in the `p->thread_info->cpu` field.

2. Checks if the state of the process `p->state` belongs to the mask of states `state` passed as argument to the function; if this is not the case, it jumps to step 9 to terminate the function.

3. If the `p->array` field is not `NULL`, the process already belongs to a runqueue; therefore, it jumps to step 8.

4. In multiprocessor systems, it checks whether the process to be awakened should be migrated from the runqueue of the lastly executing CPU to the runqueue of another CPU. Essentially, the function selects a target runqueue according to some heuristic rules. For example:

   - If some CPU in the system is idle, it selects its runqueue as the target. Preference is given to the previously executing CPU and to the local CPU, in this order.

   - If the workload of the previously executing CPU is significantly lower than the workload of the local CPU, it selects the old runqueue as the target.

   - If the process has been executed recently, it selects the old runqueue as the target (the hardware cache might still be filled with the data of the process).

   - If moving the process to the local CPU reduces the unbalance between the CPUs, the target is the local runqueue (see the section "Runqueue Balancing in Multiprocessor Systems" later in this chapter).

   After this step has been executed, the function has identified a target CPU that will execute the awakened process and, correspondingly, a target runqueue `rq` in which to insert the process.

5. If the process is in the `TASK_UNINTERRUPTIBLE` state, it decreases the `nr_uninterruptible` field of the target runqueue, and sets the `p->activated` field of the process descriptor to -1. See the later section "The recalc_task_prio() Function" for an explanation of the `activated` field.

6. Invokes the `activate_task()` function, which in turn performs the following substeps:

   a. Invokes `sched_clock()` to get the current timestamp in nanoseconds. If the target CPU is not the local CPU, it compensates for the drift of the local timer interrupts by using the timestamps relative to the last occurrences of the timer interrupts on the local and target CPUs:

   ```
   now = (sched_clock() - this_rq()->timestamp_last_tick)
       + rq->timestamp_last_tick;
   ```

b. Invokes `recalc_task_prio()`, passing to it the process descriptor pointer and the timestamp computed in the previous step. The `recalc_task_prio()` function is described in the next section.

c. Sets the value of the `p->activated` field according to Table 7-6 later in this chapter.

d. Sets the `p->timestamp` field with the timestamp computed in step 6a.

e. Inserts the process descriptor in the active set:

```
enqueue_task(p, rq->active);
rq->nr_running++;
```

7. If either the target CPU is not the local CPU or if the sync flag is not set, it checks whether the new runnable process has a dynamic priority higher than that of the current process of the `rq` runqueue (`p->prio < rq->curr->prio`); if so, invokes `resched_task()` to preempt `rq->curr`. In uniprocessor systems the latter function just executes `set_tsk_need_resched()` to set the `TIF_NEED_RESCHED` flag of the `rq->curr` process. In multiprocessor systems `resched_task()` also checks whether the old value of whether `TIF_NEED_RESCHED` flag was zero, the target CPU is different from the local CPU, and whether the `TIF_POLLING_NRFLAG` flag of the `rq->curr` process is clear (the target CPU is not actively polling the status of the `TIF_NEED_RESCHED` flag of the process). If so, `resched_task()` invokes `smp_send_reschedule()` to raise an IPI and force rescheduling on the target CPU (see the section "Interprocessor Interrupt Handling" in Chapter 4).

8. Sets the `p->state` field of the process to `TASK_RUNNING`.

9. Invokes `task_rq_unlock()` to unlock the `rq` runqueue and reenable the local interrupts.

10. Returns 1 (if the process has been successfully awakened) or 0 (if the process has not been awakened).

The recalc_task_prio() Function

The `recalc_task_prio()` function updates the average sleep time and the dynamic priority of a process. It receives as its parameters a process descriptor pointer `p` and a timestamp `now` computed by the `sched_clock()` function.

The function executes the following operations:

1. Stores in the `sleep_time` local variable the result of:

$$\min{(now - p\text{->}timestamp, 10^9)}$$

The `p->timestamp` field contains the timestamp of the process switch that put the process to sleep; therefore, `sleep_time` stores the number of nanoseconds that the process spent sleeping since its last execution (or the equivalent of 1 second, if the process slept more).

2. If `sleep_time` is not greater than zero, it jumps to step 8 so as to skip updating the average sleep time of the process.

3. Checks whether the process is not a kernel thread, whether it is awakening from the `TASK_UNINTERRUPTIBLE` state (`p->activated` field equal to −1; see step 5 in the previous section), and whether it has been continuously asleep beyond a given sleep time threshold. If these three conditions are fulfilled, the function sets the `p->sleep_avg` field to the equivalent of 900 ticks (an empirical value obtained by subtracting the duration of the base time quantum of a standard process from the maximum average sleep time). Then, it jumps to step 8.

    The sleep time threshold depends on the static priority of the process; some typical values are shown in Table 7-2. In short, the goal of this empirical rule is to ensure that processes that have been asleep for a long time in uninterruptible mode—usually waiting for disk I/O operations—get a predefined sleep average value that is large enough to allow them to be quickly serviced, but it is also not so large to cause starvation for other processes.

4. Executes the `CURRENT_BONUS` macro to compute the *bonus* value of the previous average sleep time of the process (see Table 7-3). If (10 − *bonus*) is greater than zero, the function multiplies `sleep_time` by this value. Since `sleep_time` will be added to the average sleep time of the process (see step 6 below), the lower the current average sleep time is, the more rapidly it will rise.

5. If the process is in `TASK_UNINTERRUPTIBLE` mode and it is not a kernel thread, it performs the following substeps:

    a. Checks whether the average sleep time `p->sleep_avg` is greater than or equal to its sleep time threshold (see Table 7-2 earlier in this chapter). If so, it resets the `sleep_time` local variable to zero—thus skipping the adjustment of the average sleep time—and jumps to step 6.

    b. If the sum `sleep_time`+`p->sleep_avg` is greater than or equal to the sleep time threshold, it sets the `p->sleep_avg` field to the sleep time threshold, and sets `sleep_time` to zero.

    By somewhat limiting the increment of the average sleep time of the process, the function does not reward too much batch processes that sleep for a long time.

6. Adds `sleep_time` to the average sleep time of the process (`p->sleep_avg`).

7. Checks whether `p->sleep_avg` exceeds 1000 ticks (in nanoseconds); if so, the function cuts it down to 1000 ticks (in nanoseconds).

8. Updates the dynamic priority of the process:

    ```
    p->prio = effective_prio(p);
    ```

    The `effective_prio()` function has already been discussed in the section "The scheduler_tick() Function" earlier in this chapter.

The schedule( ) Function

The schedule( ) function implements the scheduler. Its objective is to find a process in the runqueue list and then assign the CPU to it. It is invoked, directly or in a lazy (deferred) way, by several kernel routines.

Direct invocation

The scheduler is invoked directly when the current process must be blocked right away because the resource it needs is not available. In this case, the kernel routine that wants to block it proceeds as follows:

1. Inserts current in the proper wait queue.
2. Changes the state of current either to TASK_INTERRUPTIBLE or to TASK_UNINTERRUPTIBLE.
3. Invokes schedule( ).
4. Checks whether the resource is available; if not, goes to step 2.
5. Once the resource is available, removes current from the wait queue.

The kernel routine checks repeatedly whether the resource needed by the process is available; if not, it yields the CPU to some other process by invoking schedule( ). Later, when the scheduler once again grants the CPU to the process, the availability of the resource is rechecked. These steps are similar to those performed by wait_event( ) and similar functions described in the section "How Processes Are Organized" in Chapter 3.

The scheduler is also directly invoked by many device drivers that execute long iterative tasks. At each iteration cycle, the driver checks the value of the TIF_NEED_RESCHED flag and, if necessary, invokes schedule( ) to voluntarily relinquish the CPU.

Lazy invocation

The scheduler can also be invoked in a lazy way by setting the TIF_NEED_RESCHED flag of current to 1. Because a check on the value of this flag is always made before resuming the execution of a User Mode process (see the section "Returning from Interrupts and Exceptions" in Chapter 4), schedule( ) will definitely be invoked at some time in the near future.

Typical examples of lazy invocation of the scheduler are:

- When current has used up its quantum of CPU time; this is done by the scheduler_tick( ) function.
- When a process is woken up and its priority is higher than that of the current process; this task is performed by the try_to_wake_up( ) function.
- When a sched_setscheduler( ) system call is issued (see the section "System Calls Related to Scheduling" later in this chapter).

Actions performed by schedule( ) before a process switch

The goal of the schedule( ) function consists of replacing the currently executing process with another one. Thus, the key outcome of the function is to set a local variable called next, so that it points to the descriptor of the process selected to replace current. If no runnable process in the system has priority greater than the priority of current, at the end, next coincides with current and no process switch takes place.

The schedule( ) function starts by disabling kernel preemption and initializing a few local variables:

```
need_resched:

preempt_disable();
prev = current;
rq = this_rq();
```

As you see, the pointer returned by current is saved in prev, and the address of the runqueue data structure corresponding to the local CPU is saved in rq.

Next, schedule( ) makes sure that prev doesn't hold the big kernel lock (see the section "The Big Kernel Lock" in Chapter 5):

```
if (prev->lock_depth >= 0)
    up(&kernel_sem);
```

Notice that schedule( ) doesn't change the value of the lock_depth field; when prev resumes its execution, it reacquires the kernel_flag spin lock if the value of this field is not negative. Thus, the big kernel lock is automatically released and reacquired across a process switch.

The sched_clock( ) function is invoked to read the TSC and convert its value to nanoseconds; the timestamp obtained is saved in the now local variable. Then, schedule( ) computes the duration of the CPU time slice used by prev:

```
now = sched_clock();
run_time = now - prev->timestamp;
if (run_time > 1000000000)
    run_time = 1000000000;
```

The usual cut-off at 1 second (converted to nanoseconds) applies. The run_time value is used to charge the process for the CPU usage. However, a process having a high average sleep time is favored:

```
run_time /= (CURRENT_BONUS(prev) ? : 1);
```

Remember that CURRENT_BONUS returns a value between 0 and 10 that is proportional to the average sleep time of the process.

Before starting to look at the runnable processes, schedule( ) must disable the local interrupts and acquire the spin lock that protects the runqueue:

```
spin_lock_irq(&rq->lock);
```

As explained in the section "Process Termination" in Chapter 3, prev might be a process that is being terminated. To recognize this case, schedule( ) looks at the PF_DEAD flag:

```
if (prev->flags & PF_DEAD)
    prev->state = EXIT_DEAD;
```

Next, schedule( ) examines the state of prev. If it is not runnable and it has not been preempted in Kernel Mode (see the section "Returning from Interrupts and Exceptions" in Chapter 4), then it should be removed from the runqueue. However, if it has nonblocked pending signals and its state is TASK_INTERRUPTIBLE, the function sets the process state to TASK_RUNNING and leaves it into the runqueue. This action is not the same as assigning the processor to prev; it just gives prev a chance to be selected for execution:

```
if (prev->state != TASK_RUNNING &&
    !(preempt_count() & PREEMPT_ACTIVE)) {
    if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
        prev->state = TASK_RUNNING;
    else {
        if (prev->state == TASK_UNINTERRUPTIBLE)
            rq->nr_uninterruptible++;
        deactivate_task(prev, rq);
    }
}
```

The deactivate_task( ) function removes the process from the runqueue:

```
rq->nr_running--;
dequeue_task(p, p->array);
p->array = NULL;
```

Now, schedule() checks the number of runnable processes left in the runqueue. If there are some runnable processes, the function invokes the dependent_sleeper() function. In most cases, this function immediately returns zero. If, however, the kernel supports the hyper-threading technology (see the section "Runqueue Balancing in Multiprocessor Systems" later in this chapter), the function checks whether the process that is going to be selected for execution has significantly lower priority than a sibling process already running on a logical CPU of the same physical CPU; in this particular case, schedule() refuses to select the lower privilege process and executes the *swapper* process instead.

```
if (rq->nr_running) {
    if (dependent_sleeper(smp_processor_id(), rq)) {
        next = rq->idle;
        goto switch_tasks;
    }
}
```

If no runnable process exists, the function invokes idle_balance() to move some runnable process from another runqueue to the local runqueue; idle_balance() is simi-

lar to load_balance(), which is described in the later section "The load_balance()
Function."

```
if (!rq->nr_running) {
    idle_balance(smp_processor_id(), rq);
    if (!rq->nr_running) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        wake_sleeping_dependent(smp_processor_id(), rq);
        if (!rq->nr_running)
            goto switch_tasks;
    }
}
```

If idle_balance() fails in moving some process in the local runqueue, schedule()
invokes wake_sleeping_dependent() to reschedule runnable processes in *idle CPUs*
(that is, in every CPU that runs the *swapper* process). As explained earlier when dis-
cussing the dependent_sleeper() function, this unusual case might happen when the
kernel supports the hyper-threading technology. However, in uniprocessor systems,
or when all attempts to move a runnable process in the local runqueue have failed,
the function picks the *swapper* process as next and continues with the next phase.

Let's suppose that the schedule() function has determined that the runqueue
includes some runnable processes; now it has to check that at least one of these run-
nable processes is active. If not, the function exchanges the contents of the active and
expired fields of the runqueue data structure; thus, all expired processes become
active, while the empty set is ready to receive the processes that will expire in the
future.

```
array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = 140;
}
```

It is time to look up a runnable process in the active prio_array_t data structure (see
the section "Identifying a Process" in Chapter 3). First of all, schedule() searches for
the first nonzero bit in the bitmask of the active set. Remember that a bit in the bit-
mask is set when the corresponding priority list is not empty. Thus, the index of the
first nonzero bit indicates the list containing the best process to run. Then, the first
process descriptor in that list is retrieved:

```
idx = sched_find_first_bit(array->bitmap);
next = list_entry(array->queue[idx].next, task_t, run_list);
```

The sched_find_first_bit() function is based on the bsfl assembly language
instruction, which returns the bit index of the least significant bit set to one in a 32-
bit word.

The next local variable now stores the descriptor pointer of the process that will replace prev. The schedule() function looks at the next->activated field. This field encodes the state of the process when it was awakened, as illustrated in Table 7-6.

Table 7-6. The meaning of the activated field in the process descriptor

Value	Description
0	The process was in TASK_RUNNING state.
1	The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by a system call service routine or a kernel thread.
2	The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by an interrupt handler or a deferrable function.
−1	The process was in TASK_UNINTERRUPTIBLE state and it is being awakened.

If next is a conventional process and it is being awakened from the TASK_INTERRUPTIBLE or TASK_STOPPED state, the scheduler adds to the average sleep time of the process the nanoseconds elapsed since the process was inserted into the runqueue. In other words, the sleep time of the process is increased to cover also the time spent by the process in the runqueue waiting for the CPU:

```
if (next->prio >= 100 && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;
    if (next->activated == 1)
        delta = (delta * 38) / 128;
    array = next->array;
    dequeue_task(next, array);
    recalc_task_prio(next, next->timestamp + delta);
    enqueue_task(next, array);
}
next->activated = 0;
```

Observe that the scheduler makes a distinction between a process awakened by an interrupt handler or deferrable function, and a process awakened by a system call service routine or a kernel thread. In the former case, the scheduler adds the whole runqueue waiting time, while in the latter it adds just a fraction of that time. This is because interactive processes are more likely to be awakened by asynchronous events (think of the user pressing keys on the keyboard) rather than by synchronous ones.

Actions performed by schedule() to make the process switch

Now the schedule() function has determined the next process to run. In a moment, the kernel will access the thread_info data structure of next, whose address is stored close to the top of next's process descriptor:

```
switch_tasks:

prefetch(next);
```

The prefetch macro is a hint to the CPU control unit to bring the contents of the first fields of next's process descriptor in the hardware cache. It is here just to improve the

performance of `schedule()`, because the data are moved in parallel to the execution of the following instructions, which do not affect `next`.

Before replacing `prev`, the scheduler should do some administrative work:

```
clear_tsk_need_resched(prev);
rcu_qsctr_inc(prev->thread_info->cpu);
```

The `clear_tsk_need_resched()` function clears the `TIF_NEED_RESCHED` flag of `prev`, just in case `schedule()` has been invoked in the lazy way. Then, the function records that the CPU is going through a quiescent state (see the section "Read-Copy Update (RCU)" in Chapter 5).

The `schedule()` function must also decrease the average sleep time of `prev`, charging to it the slice of CPU time used by the process:

```
prev->sleep_avg -= run_time;
if ((long)prev->sleep_avg <= 0)
    prev->sleep_avg = 0;
prev->timestamp = prev->last_ran = now;
```

The timestamps of the process are then updated.

It is quite possible that `prev` and `next` are the same process: this happens if no other higher or equal priority active process is present in the runqueue. In this case, the function skips the process switch:

```
if (prev == next) {
    spin_unlock_irq(&rq->lock);
    goto finish_schedule;
}
```

At this point, `prev` and `next` are different processes, and the process switch is for real:

```
next->timestamp = now;
rq->nr_switches++;
rq->curr = next;
prev = context_switch(rq, prev, next);
```

The `context_switch()` function sets up the address space of `next`. As we'll see in "Memory Descriptor of Kernel Threads" in Chapter 9, the `active_mm` field of the process descriptor points to the memory descriptor that is used by the process, while the `mm` field points to the memory descriptor owned by the process. For normal processes, the two fields hold the same address; however, a kernel thread does not have its own address space and its `mm` field is always set to `NULL`. The `context_switch()` function ensures that if `next` is a kernel thread, it uses the address space used by `prev`:

```
if (!next->mm) {
    next->active_mm = prev->active_mm;
    atomic_inc(&prev->active_mm->mm_count);
    enter_lazy_tlb(prev->active_mm, next);
}
```

Up to Linux version 2.2, kernel threads had their own address space. That design choice was suboptimal, because the Page Tables had to be changed whenever the

scheduler selected a new process, even if it was a kernel thread. Because kernel threads run in Kernel Mode, they use only the fourth gigabyte of the linear address space, whose mapping is the same for all processes in the system. Even worse, writing into the `cr3` register invalidates all TLB entries (see "Translation Lookaside Buffers (TLB)" in Chapter 2), which leads to a significant performance penalty. Linux is nowadays much more efficient because Page Tables aren't touched at all if `next` is a kernel thread. As further optimization, if `next` is a kernel thread, the `schedule()` function sets the process into lazy TLB mode (see the section "Translation Lookaside Buffers (TLB)" in Chapter 2).

Conversely, if `next` is a regular process, the `context_switch()` function replaces the address space of `prev` with the one of `next`:

```
if (next->mm)
    switch_mm(prev->active_mm, next->mm, next);
```

If `prev` is a kernel thread or an exiting process, the `context_switch()` function saves the pointer to the memory descriptor used by `prev` in the runqueue's `prev_mm` field, then resets `prev->active_mm`:

```
if (!prev->mm) {
    rq->prev_mm = prev->active_mm;
    prev->active_mm = NULL;
}
```

Now `context_switch()` can finally invoke `switch_to()` to perform the process switch between `prev` and `next` (see the section "Performing the Process Switch" in Chapter 3):

```
switch_to(prev, next, prev);
return prev;
```

Actions performed by schedule( ) after a process switch

The instructions of the `context_switch()` and `schedule()` functions following the `switch_to` macro invocation will not be performed right away by the `next` process, but at a later time by `prev` when the scheduler selects it again for execution. However, at that moment, the `prev` local variable does not point to our original process that was to be replaced when we started the description of `schedule( )`, but rather to the process that was replaced by our original `prev` when it was scheduled again. (If you are confused, go back and read the section "Performing the Process Switch" in Chapter 3.) The first instructions after a process switch are:

```
barrier();
finish_task_switch(prev);
```

Right after the invocation of the `context_switch()` function in `schedule()`, the `barrier()` macro yields an optimization barrier for the code (see the section "Optimization and Memory Barriers" in Chapter 5). Then, the `finish_task_switch()` function is executed:

```
mm = this_rq()->prev_mm;
this_rq()->prev_mm = NULL;
```

```
    prev_task_flags = prev->flags;
    spin_unlock_irq(&this_rq()->lock);
    if (mm)
        mmdrop(mm);
    if (prev_task_flags & PF_DEAD)
        put_task_struct(prev);
```

If prev is a kernel thread, the prev_mm field of the runqueue stores the address of the memory descriptor that was lent to prev. As we'll see in Chapter 9, mmdrop() decreases the usage counter of the memory descriptor; if the counter reaches 0 (likely because prev is a zombie process), the function also frees the descriptor together with the associated Page Tables and virtual memory regions.

The finish_task_switch() function also releases the spin lock of the runqueue and enables the local interrupts. Then, it checks whether prev is a zombie task that is being removed from the system (see the section "Process Termination" in Chapter 3); if so, it invokes put_task_struct() to free the process descriptor reference counter and drop all remaining references to the process (see the section "Process Removal" in Chapter 3).

The very last instructions of the schedule() function are:

```
    finish_schedule:

    prev = current;
    if (prev->lock_depth >= 0)
        __reacquire_kernel_lock();
    preempt_enable_no_resched();
    if (test_bit(TIF_NEED_RESCHED, &current_thread_info()->flags)
        goto need_resched;
    return;
```

As you see, schedule() reacquires the big kernel lock if necessary, reenables kernel preemption, and checks whether some other process has set the TIF_NEED_RESCHED flag of the current process. In this case, the whole schedule() function is reexecuted from the beginning; otherwise, the function terminates.

Runqueue Balancing in Multiprocessor Systems

We have seen in Chapter 4 that Linux sticks to the Symmetric Multiprocessing model (SMP); this means, essentially, that the kernel should not have any bias toward one CPU with respect to the others. However, multiprocessor machines come in many different flavors, and the scheduler behaves differently depending on the hardware characteristics. In particular, we will consider the following three types of multiprocessor machines:

Classic multiprocessor architecture
> Until recently, this was the most common architecture for multiprocessor machines. These machines have a common set of RAM chips shared by all CPUs.

Hyper-threading

A hyper-threaded chip is a microprocessor that executes several threads of execution at once; it includes several copies of the internal registers and quickly switches between them. This technology, which was invented by Intel, allows the processor to exploit the machine cycles to execute another thread while the current thread is stalled for a memory access. A hyper-threaded physical CPU is seen by Linux as several different logical CPUs.

NUMA

CPUs and RAM chips are grouped in local "nodes" (usually a node includes one CPU and a few RAM chips). The memory arbiter (a special circuit that serializes the accesses to RAM performed by the CPUs in the system, see the section "Memory Addresses" in Chapter 2) is a bottleneck for the performance of the classic multiprocessor systems. In a NUMA architecture, when a CPU accesses a "local" RAM chip inside its own node, there is little or no contention, thus the access is usually fast; on the other hand, accessing a "remote" RAM chip outside of its node is much slower. We'll mention in the section "Non-Uniform Memory Access (NUMA)" in Chapter 8 how the Linux kernel memory allocator supports NUMA architectures.

These basic kinds of multiprocessor systems are often combined. For instance, a motherboard that includes two different hyper-threaded CPUs is seen by the kernel as four logical CPUs.

As we have seen in the previous section, the schedule( ) function picks the new process to run from the runqueue of the local CPU. Therefore, a given CPU can execute only the runnable processes that are contained in the corresponding runqueue. On the other hand, a runnable process is always stored in exactly one runqueue: no runnable process ever appears in two or more runqueues. Therefore, until a process remains runnable, it is usually bound to one CPU.

This design choice is usually beneficial for system performance, because the hardware cache of every CPU is likely to be filled with data owned by the runnable processes in the runqueue. In some cases, however, binding a runnable process to a given CPU might induce a severe performance penalty. For instance, consider a large number of batch processes that make heavy use of the CPU: if most of them end up in the same runqueue, one CPU in the system will be overloaded, while the others will be nearly idle.

Therefore, the kernel periodically checks whether the workloads of the runqueues are balanced and, if necessary, moves some process from one runqueue to another. However, to get the best performance from a multiprocessor system, the load balancing algorithm should take into consideration the topology of the CPUs in the system. Starting from kernel version 2.6.7, Linux sports a sophisticated runqueue balancing algorithm based on the notion of "scheduling domains." Thanks to the scheduling domains, the algorithm can be easily tuned for all kinds of existing multiprocessor

architectures (and even for recent architectures such as those based on the "multi-core" microprocessors).

Scheduling Domains

Essentially, a *scheduling domain* is a set of CPUs whose workloads should be kept balanced by the kernel. Generally speaking, scheduling domains are hierarchically organized: the top-most scheduling domain, which usually spans all CPUs in the system, includes children scheduling domains, each of which include a subset of the CPUs. Thanks to the hierarchy of scheduling domains, workload balancing can be done in a rather efficient way.

Every scheduling domain is partitioned, in turn, in one or more *groups*, each of which represents a subset of the CPUs of the scheduling domain. Workload balancing is always done between groups of a scheduling domain. In other words, a process is moved from one CPU to another only if the total workload of some group in some scheduling domain is significantly lower than the workload of another group in the same scheduling domain.

Figure 7-2 illustrates three examples of scheduling domain hierarchies, corresponding to the three main architectures of multiprocessor machines.
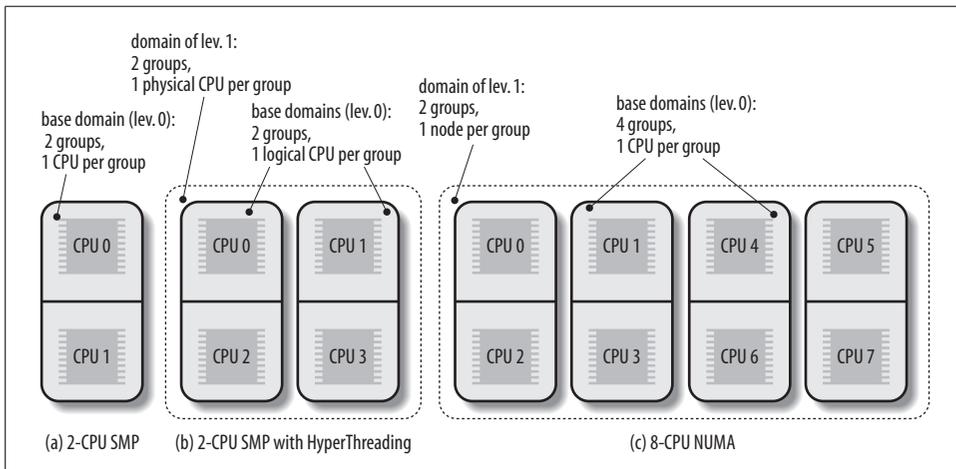
Figure 7-2. Three examples of scheduling domain hierarchies

Figure 7-2 (a) represents a hierarchy composed by a single scheduling domain for a 2-CPU classic multiprocessor architecture. The scheduling domain includes only two groups, each of which includes one CPU.

Figure 7-2 (b) represents a two-level hierarchy for a 2-CPU multiprocessor box with hyper-threading technology. The top-level scheduling domain spans all four logical CPUs in the system, and it is composed by two groups. Each group of the top-level

---

domain corresponds to a child scheduling domain and spans a physical CPU. The bottom-level scheduling domains (also called *base scheduling domains*) include two groups, one for each logical CPU.

Finally, Figure 7-2 (c) represents a two-level hierarchy for an 8-CPU NUMA architecture with two nodes and four CPUs per node. The top-level domain is organized in two groups, each of which corresponds to a different node. Every base scheduling domain spans the CPUs inside a single node and has four groups, each of which spans a single CPU.

Every scheduling domain is represented by a `sched_domain` descriptor, while every group inside a scheduling domain is represented by a `sched_group` descriptor. Each `sched_domain` descriptor includes a field `groups`, which points to the first element in a list of group descriptors. Moreover, the `parent` field of the `sched_domain` structure points to the descriptor of the parent scheduling domain, if any.

The `sched_domain` descriptors of all physical CPUs in the system are stored in the per-CPU variable `phys_domains`. If the kernel does not support the hyper-threading technology, these domains are at the bottom level of the domain hierarchy, and the `sd` fields of the runqueue descriptors point to them—that is, they are the base scheduling domains. Conversely, if the kernel supports the hyper-threading technology, the bottom-level scheduling domains are stored in the per-CPU variable `cpu_domains`.

The rebalance_tick( ) Function

To keep the runqueues in the system balanced, the `rebalance_tick()` function is invoked by `scheduler_tick()` once every tick. It receives as its parameters the index `this_cpu` of the local CPU, the address `this_rq` of the local runqueue, and a flag, `idle`, which can assume the following values:

SCHED_IDLE
> The CPU is currently idle, that is, `current` is the *swapper* process.

NOT_IDLE
> The CPU is not currently idle, that is, `current` is not the *swapper* process.

The `rebalance_tick()` function determines first the number of processes in the runqueue and updates the runqueue's average workload; to do this, the function accesses the `nr_running` and `cpu_load` fields of the runqueue descriptor.

Then, `rebalance_tick()` starts a loop over all scheduling domains in the path from the base domain (referenced by the `sd` field of the local runqueue descriptor) to the top-level domain. In each iteration the function determines whether the time has come to invoke the `load_balance()` function, thus executing a rebalancing operation on the scheduling domain. The value of `idle` and some parameters stored in the `sched_domain` descriptor determine the frequency of the invocations of `load_balance()`. If `idle` is equal to `SCHED_IDLE`, then the runqueue is empty, and `rebalance_tick()`

invokes `load_balance()` quite often (roughly once every one or two ticks for scheduling domains corresponding to logical and physical CPUs). Conversely, if `idle` is equal to `NOT_IDLE`, `rebalance_tick()` invokes `load_balance()` sparingly (roughly once every 10 milliseconds for scheduling domains corresponding to logical CPUs, and once every 100 milliseconds for scheduling domains corresponding to physical CPUs).

The load_balance( ) Function

The `load_balance()` function checks whether a scheduling domain is significantly unbalanced; more precisely, it checks whether unbalancing can be reduced by moving some processes from the busiest group to the runqueue of the local CPU. If so, the function attempts this migration. It receives four parameters:

`this_cpu`
> The index of the local CPU

`this_rq`
> The address of the descriptor of the local runqueue

`sd`
> Points to the descriptor of the scheduling domain to be checked

`idle`
> Either `SCHED_IDLE` (local CPU is idle) or `NOT_IDLE`

The function performs the following operations:

1. Acquires the `this_rq->lock` spin lock.

2. Invokes the `find_busiest_group()` function to analyze the workloads of the groups inside the scheduling domain. The function returns the address of the `sched_group` descriptor of the busiest group, provided that this group does not include the local CPU; in this case, the function also returns the number of processes to be moved into the local runqueue to restore balancing. On the other hand, if either the busiest group includes the local CPU or all groups are essentially balanced, the function returns `NULL`. This procedure is not trivial, because the function tries to filter the statistical fluctuations in the workloads.

3. If `find_busiest_group()` did not find a group not including the local CPU that is significantly busier than the other groups in the scheduling domain, the function releases the `this_rq->lock` spin lock, tunes the parameters in the scheduling domain descriptor so as to delay the next invocation of `load_balance()` on the local CPU, and terminates.

4. Invokes the `find_busiest_queue()` function to find the busiest CPUs in the group found in step 2. The function returns the descriptor address `busiest` of the corresponding runqueue.

5. Acquires a second spin lock, namely the `busiest->lock` spin lock. To prevent deadlocks, this has to be done carefully: the `this_rq->lock` is first released, then the two locks are acquired by increasing CPU indices.

6. Invokes the `move_tasks()` function to try moving some processes from the busiest runqueue to the local runqueue `this_rq` (see the next section).

7. If the `move_task()` function failed in migrating some process to the local runqueue, the scheduling domain is still unbalanced. Sets to 1 the `busiest->active_balance` flag and wakes up the *migration* kernel thread whose descriptor is stored in `busiest->migration_thread`. The *migration* kernel thread walks the chain of the scheduling domain, from the base domain of the `busiest` runqueue to the top domain, looking for an idle CPU. If an idle CPU is found, the kernel thread invokes `move_tasks()` to move one process into the idle runqueue.

8. Releases the `busiest->lock` and `this_rq->lock` spin locks.

9. Terminates.

The move_tasks( ) Function

The `move_tasks()` function moves processes from a source runqueue to the local runqueue. It receives six parameters: `this_rq` and `this_cpu` (the local runqueue descriptor and the local CPU index), `busiest` (the source runqueue descriptor), `max_nr_move` (the maximum number of processes to be moved), `sd` (the address of the scheduling domain descriptor in which this balancing operation is carried on), and the `idle` flag (beside `SCHED_IDLE` and `NOT_IDLE`, this flag can also be set to `NEWLY_IDLE` when the function is indirectly invoked by `idle_balance()`; see the section "The schedule( ) Function" earlier in this chapter).

The function first analyzes the expired processes of the `busiest` runqueue, starting from the higher priority ones. When all expired processes have been scanned, the function scans the active processes of the `busiest` runqueue. For each candidate process, the function invokes `can_migrate_task()`, which returns 1 if all the following conditions hold:

- The process is not being currently executed by the remote CPU.
- The local CPU is included in the `cpus_allowed` bitmask of the process descriptor.
- At least one of the following holds:
  - The local CPU is idle. If the kernel supports the hyper-threading technology, all logical CPUs in the local physical chip must be idle.
  - The kernel is having trouble in balancing the scheduling domain, because repeated attempts to move processes have failed.
  - The process to be moved is not "cache hot" (it has not recently executed on the remote CPU, so one can assume that no data of the process is included in the hardware cache of the remote CPU).

If `can_migrate_task()` returns the value 1, `move_tasks()` invokes the `pull_task()` function to move the candidate process to the local runqueue. Essentially, `pull_task()` executes `dequeue_task()` to remove the process from the remote runqueue, then executes `enqueue_task()` to insert the process in the local runqueue, and finally,

if the process just moved has higher dynamic priority than `current`, invokes `resched_task()` to preempt the current process of the local CPU.

System Calls Related to Scheduling

Several system calls have been introduced to allow processes to change their priorities and scheduling policies. As a general rule, users are always allowed to lower the priorities of their processes. However, if they want to modify the priorities of processes belonging to some other user or if they want to increase the priorities of their own processes, they must have superuser privileges.

The nice( ) System Call

The `nice()`* system call allows processes to change their base priority. The integer value contained in the `increment` parameter is used to modify the `nice` field of the process descriptor. The *nice* Unix command, which allows users to run programs with modified scheduling priority, is based on this system call.

The `sys_nice()` service routine handles the `nice()` system call. Although the `increment` parameter may have any value, absolute values larger than 40 are trimmed down to 40. Traditionally, negative values correspond to requests for priority increments and require superuser privileges, while positive ones correspond to requests for priority decreases. In the case of a negative increment, the function invokes the `capable()` function to verify whether the process has a `CAP_SYS_NICE` capability. Moreover, the function invokes the `security_task_setnice()` security hook. We discuss that function in Chapter 20. If the user turns out to have the privilege required to change priorities, `sys_nice()` converts `current->static_prio` to the range of nice values, adds the value of `increment`, and invokes the `set_user_nice()` function. In turn, the latter function gets the local runqueue lock, updates the static priority of `current`, invokes the `resched_task()` function to allow other processes to preempt `current`, and release the runqueue lock.

The `nice()` system call is maintained for backward compatibility only; it has been replaced by the `setpriority()` system call described next.

The getpriority( ) and setpriority( ) System Calls

The `nice()` system call affects only the process that invokes it. Two other system calls, denoted as `getpriority()` and `setpriority()`, act on the base priorities of all processes in a given group. `getpriority()` returns 20 minus the lowest `nice` field value among all processes in a given group—that is, the highest priority among those

---

* Because this system call is usually invoked to lower the priority of a process, users who invoke it for their processes are "nice" to other users.

processes; `setpriority( )` sets the base priority of all processes in a given group to a given value.

The kernel implements these system calls by means of the `sys_getpriority( )` and `sys_setpriority( )` service routines. Both of them act essentially on the same group of parameters:

which
> The value that identifies the group of processes; it can assume one of the following:
>
> PRIO_PROCESS
>> Selects the processes according to their process ID (`pid` field of the process descriptor).
>
> PRIO_PGRP
>> Selects the processes according to their group ID (`pgrp` field of the process descriptor).
>
> PRIO_USER
>> Selects the processes according to their user ID (`uid` field of the process descriptor).

who
> The value of the `pid`, `pgrp`, or `uid` field (depending on the value of `which`) to be used for selecting the processes. If `who` is 0, its value is set to that of the corresponding field of the `current` process.

niceval
> The new base priority value (needed only by `sys_setpriority( )`). It should range between −20 (highest priority) and +19 (lowest priority).

As stated before, only processes with a `CAP_SYS_NICE` capability are allowed to increase their own base priority or to modify that of other processes.

As we will see in Chapter 10, system calls return a negative value only if some error occurred. For this reason, `getpriority( )` does not return a normal nice value ranging between −20 and +19, but rather a nonnegative value ranging between 1 and 40.

The sched_getaffinity( ) and sched_setaffinity( ) System Calls

The `sched_getaffinity( )` and `sched_setaffinity( )` system calls respectively return and set up the CPU affinity mask of a process—the bit mask of the CPUs that are allowed to execute the process. This mask is stored in the `cpus_allowed` field of the process descriptor.

The `sys_sched_getaffinity( )` system call service routine looks up the process descriptor by invoking `find_task_by_pid( )`, and then returns the value of the corresponding `cpus_allowed` field ANDed with the bitmap of the available CPUs.

The sys_sched_setaffinity() system call is a bit more complicated. Besides looking for the descriptor of the target process and updating the cpus_allowed field, this function has to check whether the process is included in a runqueue of a CPU that is no longer present in the new affinity mask. In the worst case, the process has to be moved from one runqueue to another one. To avoid problems due to deadlocks and race conditions, this job is done by the *migration* kernel threads (there is one thread per CPU). Whenever a process has to be moved from a runqueue rq1 to another runqueue rq2, the system call awakes the migration thread of rq1 (rq1->migration_thread), which in turn removes the process from rq1 and inserts it into rq2.

System Calls Related to Real-Time Processes

We now introduce a group of system calls that allow processes to change their scheduling discipline and, in particular, to become real-time processes. As usual, a process must have a CAP_SYS_NICE capability to modify the values of the rt_priority and policy process descriptor fields of any process, including itself.

The sched_getscheduler( ) and sched_setscheduler( ) system calls

The sched_getscheduler() system call queries the scheduling policy currently applied to the process identified by the pid parameter. If pid equals 0, the policy of the calling process is retrieved. On success, the system call returns the policy for the process: SCHED_FIFO, SCHED_RR, or SCHED_NORMAL (the latter is also called SCHED_OTHER). The corresponding sys_sched_getscheduler() service routine invokes find_process_by_pid(), which locates the process descriptor corresponding to the given pid and returns the value of its policy field.

The sched_setscheduler() system call sets both the scheduling policy and the associated parameters for the process identified by the parameter pid. If pid is equal to 0, the scheduler parameters of the calling process will be set.

The corresponding sys_sched_setscheduler() system call service routine simply invokes do_sched_setscheduler(). The latter function checks whether the scheduling policy specified by the policy parameter and the new priority specified by the param->sched_priority parameter are valid. It also checks whether the process has CAP_SYS_NICE capability or whether its owner has superuser rights. If everything is OK, it removes the process from its runqueue (if it is runnable); updates the static, real-time, and dynamic priorities of the process; inserts the process back in the runqueue; and finally invokes, if necessary, the resched_task() function to preempt the current process of the runqueue.

The sched_getparam( ) and sched_setparam( ) system calls

The sched_getparam() system call retrieves the scheduling parameters for the process identified by pid. If pid is 0, the parameters of the current process are retrieved.

The corresponding sys_sched_getparam( ) service routine, as one would expect, finds the process descriptor pointer associated with pid, stores its rt_priority field in a local variable of type sched_param, and invokes copy_to_user( ) to copy it into the process address space at the address specified by the param parameter.

The sched_setparam( ) system call is similar to sched_setscheduler( ). The difference is that sched_setparam( ) does not let the caller set the policy field's value.* The corresponding sys_sched_setparam( ) service routine invokes do_sched_setscheduler(), with almost the same parameters as sys_sched_setscheduler( ).

The sched_yield( ) system call

The sched_yield( ) system call allows a process to relinquish the CPU voluntarily without being suspended; the process remains in a TASK_RUNNING state, but the scheduler puts it either in the expired set of the runqueue (if the process is a conventional one), or at the end of the runqueue list (if the process is a real-time one). The schedule( ) function is then invoked. In this way, other processes that have the same dynamic priority have a chance to run. The call is used mainly by SCHED_FIFO real-time processes.

The sched_get_priority_min( ) and sched_get_priority_max( ) system calls

The sched_get_priority_min( ) and sched_get_priority_max( ) system calls return, respectively, the minimum and the maximum real-time static priority value that can be used with the scheduling policy identified by the policy parameter.

The sys_sched_get_priority_min( ) service routine returns 1 if current is a real-time process, 0 otherwise.

The sys_sched_get_priority_max( ) service routine returns 99 (the highest priority) if current is a real-time process, 0 otherwise.

The sched_rr_get_interval( ) system call

The sched_rr_get_interval( ) system call writes into a structure stored in the User Mode address space the Round Robin time quantum for the real-time process identified by the pid parameter. If pid is zero, the system call writes the time quantum of the current process.

The corresponding sys_sched_rr_get_interval( ) service routine invokes, as usual, find_process_by_pid() to retrieve the process descriptor associated with pid. It then converts the base time quantum of the selected process into seconds and nanoseconds and copies the numbers into the User Mode structure. Conventionally, the time quantum of a FIFO real-time process is equal to zero.

---

* This anomaly is caused by a specific requirement of the POSIX standard.

CHAPTER 8
Memory Management

We saw in Chapter 2 how Linux takes advantage of 80×86's segmentation and paging circuits to translate logical addresses into physical ones. We also mentioned that some portion of RAM is permanently assigned to the kernel and used to store both the kernel code and the static kernel data structures.

The remaining part of the RAM is called *dynamic memory*. It is a valuable resource, needed not only by the processes but also by the kernel itself. In fact, the performance of the entire system depends on how efficiently dynamic memory is managed. Therefore, all current multitasking operating systems try to optimize the use of dynamic memory, assigning it only when it is needed and freeing it as soon as possible. Figure 8-1 shows schematically the page frames used as dynamic memory; see the section "Physical Memory Layout" in Chapter 2 for details.

This chapter, which consists of three main sections, describes how the kernel allocates dynamic memory for its own use. The sections "Page Frame Management" and "Memory Area Management" illustrate two different techniques for handling physically contiguous memory areas, while the section "Noncontiguous Memory Area Management" illustrates a third technique that handles noncontiguous memory areas. In these sections we'll cover topics such as memory zones, kernel mappings, the buddy system, the slab cache, and memory pools.

Page Frame Management

We saw in the section "Paging in Hardware" in Chapter 2 how the Intel Pentium processor can use two different page frame sizes: 4 KB and 4 MB (or 2 MB if PAE is enabled—see the section "The Physical Address Extension (PAE) Paging Mechanism" in Chapter 2). Linux adopts the smaller 4 KB page frame size as the standard memory allocation unit. This makes things simpler for two reasons:

- The Page Fault exceptions issued by the paging circuitry are easily interpreted. Either the page requested exists but the process is not allowed to address it, or

the page does not exist. In the second case, the memory allocator must find a free 4 KB page frame and assign it to the process.

- Although both 4 KB and 4 MB are multiples of all disk block sizes, transfers of data between main memory and disks are in most cases more efficient when the smaller size is used.
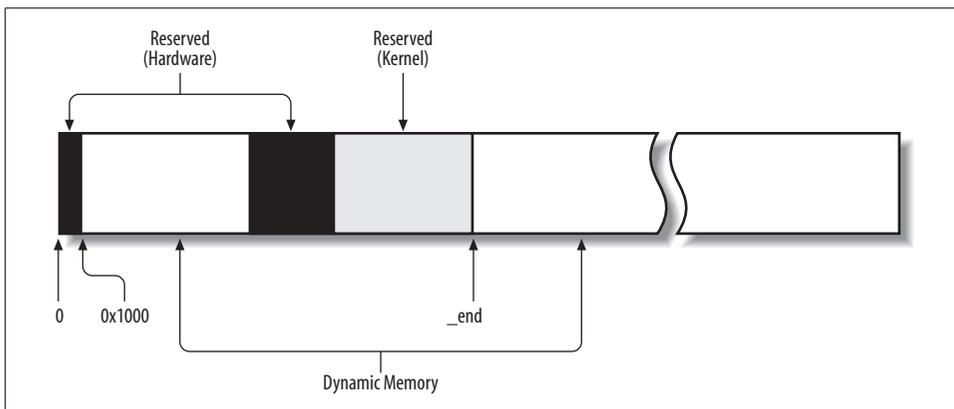
Figure 8-1. Dynamic memory

Page Descriptors

The kernel must keep track of the current status of each page frame. For instance, it must be able to distinguish the page frames that are used to contain pages that belong to processes from those that contain kernel code or kernel data structures. Similarly, it must be able to determine whether a page frame in dynamic memory is free. A page frame in dynamic memory is free if it does not contain any useful data. It is not free when the page frame contains data of a User Mode process, data of a software cache, dynamically allocated kernel data structures, buffered data of a device driver, code of a kernel module, and so on.

State information of a page frame is kept in a page descriptor of type page, whose fields are shown in Table 8-1. All page descriptors are stored in the mem_map array. Because each descriptor is 32 bytes long, the space required by mem_map is slightly less than 1% of the whole RAM. The virt_to_page(addr) macro yields the address of the page descriptor associated with the linear address addr. The pfn_to_page(pfn) macro yields the address of the page descriptor associated with the page frame having number pfn.

Table 8-1. The fields of the page descriptor

Type	Name	Description
unsigned long	flags	Array of flags (see Table 8-2). Also encodes the zone number to which the page frame belongs.
atomic_t	_count	Page frame's reference counter.

Table 8-1. The fields of the page descriptor (continued)

Type	Name	Description
`atomic_t`	`_mapcount`	Number of Page Table entries that refer to the page frame (–1 if none).
`unsigned long`	`private`	Available to the kernel component that is using the page (for instance, it is a buffer head pointer in case of buffer page; see "Block Buffers and Buffer Heads" in Chapter 15). If the page is free, this field is used by the buddy system (see later in this chapter).
`struct address_space *`	`mapping`	Used when the page is inserted into the page cache (see the section "The Page Cache" in Chapter 15), or when it belongs to an anonymous region (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
`unsigned long`	`index`	Used by several kernel components with different meanings. For instance, it identifies the position of the data stored in the page frame within the page's disk image or within an anonymous region (Chapter 15), or it stores a swapped-out page identifier (Chapter 17).
`struct list_head`	`lru`	Contains pointers to the least recently used doubly linked list of pages.

You don't have to fully understand the role of all fields in the page descriptor right now. In the following chapters, we often come back to the fields of the page descriptor. Moreover, several fields have different meaning, according to whether the page frame is free or what kernel component is using the page frame.

Let's describe in greater detail two of the fields:

`_count`

> A usage reference counter for the page. If it is set to -1, the corresponding page frame is free and can be assigned to any process or to the kernel itself. If it is set to a value greater than or equal to 0, the page frame is assigned to one or more processes or is used to store some kernel data structures. The page_count( ) function returns the value of the _count field increased by one, that is, the number of users of the page.

`flags`

> Includes up to 32 flags (see Table 8-2) that describe the status of the page frame. For each PG_*xyz* flag, the kernel defines some macros that manipulate its value. Usually, the Page*Xyz* macro returns the value of the flag, while the SetPage*Xyz* and ClearPage*Xyz* macro set and clear the corresponding bit, respectively.

Table 8-2. Flags describing the status of a page frame

Flag name	Meaning
`PG_locked`	The page is locked; for instance, it is involved in a disk I/O operation.
`PG_error`	An I/O error occurred while transferring the page.
`PG_referenced`	The page has been recently accessed.
`PG_uptodate`	This flag is set after completing a read operation, unless a disk I/O error happened.
`PG_dirty`	The page has been modified (see the section "Implementing the PFRA" in Chapter 17).

Table 8-2. Flags describing the status of a page frame (continued)

Flag name	Meaning
PG_lru	The page is in the active or inactive page list (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).
PG_active	The page is in the active page list (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).
PG_slab	The page frame is included in a slab (see the section "Memory Area Management" later in this chapter).
PG_highmem	The page frame belongs to the ZONE_HIGHMEM zone (see the following section "Non-Uniform Memory Access (NUMA)").
PG_checked	Used by some filesystems such as Ext2 and Ext3 (see Chapter 18).
PG_arch_1	Not used on the 80×86 architecture.
PG_reserved	The page frame is reserved for kernel code or is unusable.
PG_private	The private field of the page descriptor stores meaningful data.
PG_writeback	The page is being written to disk by means of the writepage method (see Chapter 16) .
PG_nosave	Used for system suspend/resume.
PG_compound	The page frame is handled through the extended paging mechanism (see the section "Extended Paging" in Chapter 2).
PG_swapcache	The page belongs to the swap cache (see the section "The Swap Cache" in Chapter 17).
PG_mappedtodisk	All data in the page frame corresponds to blocks allocated on disk.
PG_reclaim	The page has been marked to be written to disk in order to reclaim memory.
PG_nosave_free	Used for system suspend/resume.

Non-Uniform Memory Access (NUMA)

We are used to thinking of the computer's memory as a homogeneous, shared resource. Disregarding the role of the hardware caches, we expect the time required for a CPU to access a memory location to be essentially the same, regardless of the location's physical address and the CPU. Unfortunately, this assumption is not true in some architectures. For instance, it is not true for some multiprocessor Alpha or MIPS computers.

Linux 2.6 supports the *Non-Uniform Memory Access* (*NUMA*) model, in which the access times for different memory locations from a given CPU may vary. The physical memory of the system is partitioned in several *nodes*. The time needed by a given CPU to access pages within a single node is the same. However, this time might not be the same for two different CPUs. For every CPU, the kernel tries to minimize the number of accesses to costly nodes by carefully selecting where the kernel data structures that are most often referenced by the CPU are stored.[*]

---

[*] Furthermore, the Linux kernel makes use of NUMA even for some peculiar uniprocessor systems that have huge "holes" in the physical address space. The kernel handles these architectures by assigning the contiguous subranges of valid physical addresses to different memory nodes.

The physical memory inside each node can be split into several zones, as we will see in the next section. Each node has a descriptor of type pg_data_t, whose fields are shown in Table 8-3. All node descriptors are stored in a singly linked list, whose first element is pointed to by the pgdat_list variable.

Table 8-3. The fields of the node descriptor

Type	Name	Description
struct zone[]	node_zones	Array of zone descriptors of the node
struct zonelist[]	node_zonelists	Array of zonelist data structures used by the page allocator (see the later section "Memory Zones")
int	nr_zones	Number of zones in the node
struct page *	node_mem_map	Array of page descriptors of the node
struct bootmem_data *	bdata	Used in the kernel initialization phase
unsigned long	node_start_pfn	Index of the first page frame in the node
unsigned long	node_present_pages	Size of the memory node, excluding holes (in page frames)
unsigned long	node_spanned_pages	Size of the node, including holes (in page frames)
int	node_id	Identifier of the node
pg_data_t *	pgdat_next	Next item in the memory node list
wait_queue_head_t	kswapd_wait	Wait queue for the *kswapd* pageout daemon (see the section "Periodic Reclaiming" in Chapter 17)
struct task_struct *	kswapd	Pointer to the process descriptor of the *kswapd* kernel thread
int	kswapd_max_order	Logarithmic size of free blocks to be created by *kswapd*

As usual, we are mostly concerned with the 80×86 architecture. IBM-compatible PCs use the Uniform Memory Access model (UMA), thus the NUMA support is not really required. However, even if NUMA support is not compiled in the kernel, Linux makes use of a single node that includes all system physical memory. Thus, the pgdat_list variable points to a list consisting of a single element—the node 0 descriptor—stored in the contig_page_data variable.

On the 80×86 architecture, grouping the physical memory in a single node might appear useless; however, this approach makes the memory handling code more portable, because the kernel can assume that the physical memory is partitioned in one or more nodes in all architectures.[*]

---

[*] We have another example of this kind of design choice: Linux uses four levels of Page Tables even when the hardware architecture defines just two levels (see the section "Paging in Linux" in Chapter 2).

Memory Zones

In an ideal computer architecture, a page frame is a memory storage unit that can be used for anything: storing kernel and user data, buffering disk data, and so on. Every kind of page of data can be stored in a page frame, without limitations.

However, real computer architectures have hardware constraints that may limit the way page frames can be used. In particular, the Linux kernel must deal with two hardware constraints of the 80 × 86 architecture:

- The Direct Memory Access (DMA) processors for old ISA buses have a strong limitation: they are able to address only the first 16 MB of RAM.
- In modern 32-bit computers with lots of RAM, the CPU cannot directly access all physical memory because the linear address space is too small.

To cope with these two limitations, Linux 2.6 partitions the physical memory of every memory node into three *zones*. In the 80 × 86 UMA architecture the zones are:

`ZONE_DMA`
> Contains page frames of memory below 16 MB

`ZONE_NORMAL`
> Contains page frames of memory at and above 16 MB and below 896 MB

`ZONE_HIGHMEM`
> Contains page frames of memory at and above 896 MB

The `ZONE_DMA` zone includes page frames that can be used by old ISA-based devices by means of the DMA. (The section "Direct Memory Access (DMA)" in Chapter 13 gives further details on DMA.)

The `ZONE_DMA` and `ZONE_NORMAL` zones include the "normal" page frames that can be directly accessed by the kernel through the linear mapping in the fourth gigabyte of the linear address space (see the section "Kernel Page Tables" in Chapter 2). Conversely, the `ZONE_HIGHMEM` zone includes page frames that cannot be directly accessed by the kernel through the linear mapping in the fourth gigabyte of linear address space (see the section "Kernel Mappings of High-Memory Page Frames" later in this chapter). The `ZONE_HIGHMEM` zone is always empty on 64-bit architectures.

Each memory zone has its own descriptor of type zone. Its fields are shown in Table 8-4.

Table 8-4. The fields of the zone descriptor

Type	Name	Description
`unsigned long`	`free_pages`	Number of free pages in the zone.
`unsigned long`	`pages_min`	Number of reserved pages of the zone (see the section "The Pool of Reserved Page Frames" later in this chapter).

Table 8-4. The fields of the zone descriptor (continued)

Type	Name	Description
unsigned long	pages_low	Low watermark for page frame reclaiming; also used by the zone allocator as a threshold value (see the section "The Zone Allocator" later in this chapter).
unsigned long	pages_high	High watermark for page frame reclaiming; also used by the zone allocator as a threshold value.
unsigned long []	lowmem_reserve	Specifies how many page frames in each zone must be reserved for handling low-on-memory critical situations.
struct per_cpu_pageset[]	pageset	Data structure used to implement special caches of single page frames (see the section "The Per-CPU Page Frame Cache" later in this chapter).
spinlock_t	lock	Spin lock protecting the descriptor.
struct free_area []	free_area	Identifies the blocks of free page frames in the zone (see the section "The Buddy System Algorithm" later in this chapter).
spinlock_t	lru_lock	Spin lock for the active and inactive lists.
struct list head	active_list	List of active pages in the zone (see Chapter 17).
struct list head	inactive_list	List of inactive pages in the zone (see Chapter 17).
unsigned long	nr_scan_active	Number of active pages to be scanned when reclaiming memory (see the section "Low On Memory Reclaiming" in Chapter 17).
unsigned long	nr_scan_inactive	Number of inactive pages to be scanned when reclaiming memory.
unsigned long	nr_active	Number of pages in the zone's active list.
unsigned long	nr_inactive	Number of pages in the zone's inactive list.
unsigned long	pages_scanned	Counter used when doing page frame reclaiming in the zone.
int	all_unreclaimable	Flag set when the zone is full of unreclaimable pages.
int	temp_priority	Temporary zone's priority (used when doing page frame reclaiming).
int	prev_priority	Zone's priority ranging between 12 and 0 (used by the page frame reclaiming algorithm, see the section "Low On Memory Reclaiming" in Chapter 17).
wait_queue_head_t *	wait_table	Hash table of wait queues of processes waiting for one of the pages of the zone.
unsigned long	wait_table_size	Size of the wait queue hash table.
unsigned long	wait_table_bits	Power-of-2 order of the size of the wait queue hash table array.
struct pglist_data *	zone_pgdat	Memory node (see the earlier section "Non-Uniform Memory Access (NUMA)").
struct page *	zone_mem_map	Pointer to first page descriptor of the zone.
unsigned long	zone_start_pfn	Index of the first page frame of the zone.
unsigned long	spanned_pages	Total size of zone in pages, including holes.

Table 8-4. The fields of the zone descriptor (continued)

Type	Name	Description
`unsigned long`	`present_pages`	Total size of zone in pages, excluding holes.
`char *`	`name`	Pointer to the conventional name of the zone: "DMA," "Normal," or "HighMem."

Many fields of the `zone` structure are used for page frame reclaiming and will be described in Chapter 17.

Each page descriptor has links to the memory node and to the zone inside the node that includes the corresponding page frame. To save space, these links are not stored as classical pointers; rather, they are encoded as indices stored in the high bits of the `flags` field. In fact, the number of flags that characterize a page frame is limited, thus it is always possible to reserve the most significant bits of the `flags` field to encode the proper memory node and zone number.* The `page_zone()` function receives as its parameter the address of a page descriptor; it reads the most significant bits of the `flags` field in the page descriptor, then it determines the address of the corresponding zone descriptor by looking in the `zone_table` array. This array is initialized at boot time with the addresses of all zone descriptors of all memory nodes.

When the kernel invokes a memory allocation function, it must specify the zones that contain the requested page frames. The kernel usually specifies which zones it's willing to use. For instance, if a page frame must be directly mapped in the fourth gigabyte of linear addresses but it is not going to be used for ISA DMA transfers, then the kernel requests a page frame either in `ZONE_NORMAL` or in `ZONE_DMA`. Of course, the page frame should be obtained from `ZONE_DMA` only if `ZONE_NORMAL` does not have free page frames. To specify the preferred zones in a memory allocation request, the kernel uses the `zonelist` data structure, which is an array of zone descriptor pointers.

The Pool of Reserved Page Frames

Memory allocation requests can be satisfied in two different ways. If enough free memory is available, the request can be satisfied immediately. Otherwise, some memory reclaiming must take place, and the kernel control path that made the request is blocked until additional memory has been freed.

However, some kernel control paths cannot be blocked while requesting memory—this happens, for instance, when handling an interrupt or when executing code inside a critical region. In these cases, a kernel control path should issue *atomic memory*

---

* The number of bits reserved for the indices depends on whether the kernel supports the NUMA model and on the size of the `flags` field. If NUMA is not supported, the `flags` field has two bits for the zone index and one bit—always set to zero—for the node index. On NUMA 32-bit architectures, `flags` has two bits for the zone index and six bits for the node number. Finally, on NUMA 64-bit architectures, the 64-bit `flags` field has 2 bits for the zone index and 10 bits for the node number.

allocation requests (using the `GFP_ATOMIC` flag; see the later section "The Zoned Page Frame Allocator"). An atomic request never blocks: if there are not enough free pages, the allocation simply fails.

Although there is no way to ensure that an atomic memory allocation request never fails, the kernel tries hard to minimize the likelihood of this unfortunate event. In order to do this, the kernel reserves a pool of page frames for atomic memory allocation requests to be used only on low-on-memory conditions.

The amount of the reserved memory (in kilobytes) is stored in the `min_free_kbytes` variable. Its initial value is set during kernel initialization and depends on the amount of physical memory that is directly mapped in the kernel's fourth gigabyte of linear addresses—that is, it depends on the number of page frames included in the `ZONE_DMA` and `ZONE_NORMAL` memory zones:

$$reserved\ pool\ size\ =\ \left\lfloor \sqrt{16 \times directly\ mapped\ memory} \right\rfloor \quad (kilobytes)$$

However, initially `min_free_kbytes` cannot be lower than 128 and greater than 65,536.[*]

The `ZONE_DMA` and `ZONE_NORMAL` memory zones contribute to the reserved memory with a number of page frames proportional to their relative sizes. For instance, if the `ZONE_NORMAL` zone is eight times bigger than `ZONE_DMA`, seven-eighths of the page frames will be taken from `ZONE_NORMAL` and one-eighth from `ZONE_DMA`.

The `pages_min` field of the zone descriptor stores the number of reserved page frames inside the zone. As we'll see in Chapter 17, this field plays also a role for the page frame reclaiming algorithm, together with the `pages_low` and `pages_high` fields. The `pages_low` field is always set to 5/4 of the value of `pages_min`, and `pages_high` is always set to 3/2 of the value of `pages_min`.

The Zoned Page Frame Allocator

The kernel subsystem that handles the memory allocation requests for groups of contiguous page frames is called the *zoned page frame allocator*. Its main components are shown in Figure 8-2.

The component named "zone allocator" receives the requests for allocation and deallocation of dynamic memory. In the case of allocation requests, the component searches a memory zone that includes a group of contiguous page frames that can satisfy the request (see the later section "The Zone Allocator"). Inside each zone, page frames are handled by a component named "buddy system" (see the later section "The Buddy System Algorithm"). To get better system performance, a small

---

[*] The amount of reserved memory can be changed later by the system administrator either by writing in the */proc/sys/vm/min_free_kbytes* file or by issuing a suitable `sysctl()` system call.
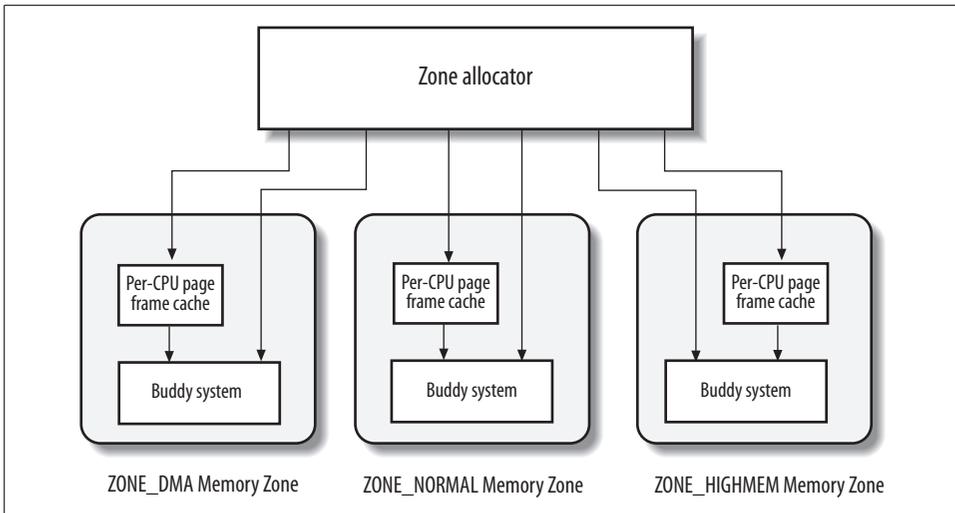
Figure 8-2. Components of the zoned page frame allocator

number of page frames are kept in cache to quickly satisfy the allocation requests for single page frames (see the later section "The Per-CPU Page Frame Cache").

Requesting and releasing page frames

Page frames can be requested by using six slightly different functions and macros. Unless otherwise stated, they return the linear address of the first allocated page or return NULL if the allocation failed.

alloc_pages(gfp_mask, order)

Macro used to request 2^{order} contiguous page frames. It returns the address of the descriptor of the first allocated page frame or returns NULL if the allocation failed.

alloc_page(gfp_mask)

Macro used to get a single page frame; it expands to:

```
alloc_pages(gfp_mask, 0)
```

It returns the address of the descriptor of the allocated page frame or returns NULL if the allocation failed.

__get_free_pages(gfp_mask, order)

Function that is similar to alloc_pages(), but it returns the linear address of the first allocated page.

__get_free_page(gfp_mask)

Macro used to get a single page frame; it expands to:

```
__get_free_pages(gfp_mask, 0)
```

```
get_zeroed_page(gfp_mask)
```
    Function used to obtain a page frame filled with zeros; it invokes:
```
        alloc_pages(gfp_mask | __GFP_ZERO, 0)
```
    and returns the linear address of the obtained page frame.

```
__get_dma_pages(gfp_mask, order)
```
    Macro used to get page frames suitable for DMA; it expands to:
```
        __get_free_pages(gfp_mask | __GFP_DMA, order)
```

The parameter `gfp_mask` is a group of flags that specify how to look for free page frames. The flags that can be used in `gfp_mask` are shown in Table 8-5.

Table 8-5. Flag used to request page frames

Flag	Description
__GFP_DMA	The page frame must belong to the ZONE_DMA memory zone. Equivalent to GFP_DMA.
__GFP_HIGHMEM	The page frame may belong to the ZONE_HIGHMEM memory zone.
__GFP_WAIT	The kernel is allowed to block the current process waiting for free page frames.
__GFP_HIGH	The kernel is allowed to access the pool of reserved page frames.
__GFP_IO	The kernel is allowed to perform I/O transfers on low memory pages in order to free page frames.
__GFP_FS	If clear, the kernel is not allowed to perform filesystem-dependent operations.
__GFP_COLD	The requested page frames may be "cold" (see the later section "The Per-CPU Page Frame Cache").
__GFP_NOWARN	A memory allocation failure will not produce a warning message.
__GFP_REPEAT	The kernel keeps retrying the memory allocation until it succeeds.
__GFP_NOFAIL	Same as __GFP_REPEAT.
__GFP_NORETRY	Do not retry a failed memory allocation.
__GFP_NO_GROW	The slab allocator does not allow a slab cache to be enlarged (see the later section "The Slab Allocator").
__GFP_COMP	The page frame belongs to an extended page (see the section "Extended Paging" in Chapter 2).
__GFP_ZERO	The page frame returned, if any, must be filled with zeros.

In practice, Linux uses the predefined combinations of flag values shown in Table 8-6; the group name is what you'll encounter as the argument of the six page frame allocation functions.

Table 8-6. Groups of flag values used to request page frames

Group name	Corresponding flags
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	__GFP_WAIT | __GFP_IO
GFP_KERNEL	__GFP_WAIT | __GFP_IO | __GFP_FS
GFP_USER	__GFP_WAIT | __GFP_IO | __GFP_FS
GFP_HIGHUSER	__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM

The `__GFP_DMA` and `__GFP_HIGHMEM` flags are called *zone modifiers*; they specify the zones searched by the kernel while looking for free page frames. The `node_zonelists` field of the `contig_page_data` node descriptor is an array of lists of zone descriptors representing the *fallback zones*: for each setting of the zone modifiers, the corresponding list includes the memory zones that could be used to satisfy the memory allocation request in case the original zone is short on page frames. In the 80×86 UMA architecture, the fallback zones are the following:

- If the `__GFP_DMA` flag is set, page frames can be taken only from the `ZONE_DMA` memory zone.

- Otherwise, if the `__GFP_HIGHMEM` flag is *not* set, page frames can be taken only from the `ZONE_NORMAL` and the `ZONE_DMA` memory zones, in order of preference.

- Otherwise (the `__GFP_HIGHMEM` flag is set), page frames can be taken from `ZONE_HIGHMEM`, `ZONE_NORMAL`, and `ZONE_DMA` memory zones, in order of preference.

Page frames can be released through each of the following four functions and macros:

`__free_pages(page, order)`
> This function checks the page descriptor pointed to by `page`; if the page frame is not reserved (i.e., if the `PG_reserved` flag is equal to 0), it decreases the `count` field of the descriptor. If `count` becomes 0, it assumes that 2^{order} contiguous page frames starting from the one corresponding to `page` are no longer used. In this case, the function releases the page frames as explained in the later section "The Zone Allocator."

`free_pages(addr, order)`
> This function is similar to `__free_pages()`, but it receives as an argument the linear address `addr` of the first page frame to be released.

`__free_page(page)`
> This macro releases the page frame having the descriptor pointed to by `page`; it expands to:
>
>   __free_pages(page, 0)

`free_page(addr)`
> This macro releases the page frame having the linear address `addr`; it expands to:
>
>   free_pages(addr, 0)

Kernel Mappings of High-Memory Page Frames

The linear address that corresponds to the end of the directly mapped physical memory, and thus to the beginning of the high memory, is stored in the `high_memory` variable, which is set to 896 MB. Page frames above the 896 MB boundary are not generally mapped in the fourth gigabyte of the kernel linear address spaces, so the kernel is unable to directly access them. This implies that each page allocator function that returns the linear address of the assigned page frame doesn't work for *high-memory* page frames, that is, for page frames in the `ZONE_HIGHMEM` memory zone.

For instance, suppose that the kernel invoked __get_free_pages(GFP_HIGHMEM,0) to allocate a page frame in high memory. If the allocator assigned a page frame in high memory, __get_free_pages() cannot return its linear address because it doesn't exist; thus, the function returns NULL. In turn, the kernel cannot use the page frame; even worse, the page frame cannot be released because the kernel has lost track of it.

This problem does not exist on 64-bit hardware platforms, because the available linear address space is much larger than the amount of RAM that can be installed—in short, the ZONE_HIGHMEM zone of these architectures is always empty. On 32-bit platforms such as the 80×86 architecture, however, Linux designers had to find some way to allow the kernel to exploit all the available RAM, up to the 64 GB supported by PAE. The approach adopted is the following:

- The allocation of high-memory page frames is done only through the alloc_pages() function and its alloc_page() shortcut. These functions do not return the linear address of the first allocated page frame, because if the page frame belongs to the high memory, such linear address simply does not exist. Instead, the functions return the linear address of the page descriptor of the first allocated page frame. These linear addresses always exist, because all page descriptors are allocated in low memory once and forever during the kernel initialization.

- Page frames in high memory that do not have a linear address cannot be accessed by the kernel. Therefore, part of the last 128 MB of the kernel linear address space is dedicated to mapping high-memory page frames. Of course, this kind of mapping is temporary, otherwise only 128 MB of high memory would be accessible. Instead, by recycling linear addresses the whole high memory can be accessed, although at different times.

The kernel uses three different mechanisms to map page frames in high memory; they are called *permanent kernel mapping*, *temporary kernel mapping*, and *noncontiguous memory allocation*. In this section, we'll cover the first two techniques; the third one is discussed in the section "Noncontiguous Memory Area Management" later in this chapter.

Establishing a permanent kernel mapping may block the current process; this happens when no free Page Table entries exist that can be used as "windows" on the page frames in high memory. Thus, a permanent kernel mapping cannot be established in interrupt handlers and deferrable functions. Conversely, establishing a temporary kernel mapping never requires blocking the current process; its drawback, however, is that very few temporary kernel mappings can be established at the same time.

A kernel control path that uses a temporary kernel mapping must ensure that no other kernel control path is using the same mapping. This implies that the kernel control path can never block, otherwise another kernel control path might use the same window to map some other high memory page.

Of course, none of these techniques allow addressing the whole RAM simulta-
neously. After all, less than 128 MB of linear address space are left for mapping the
high memory, while PAE supports systems having up to 64 GB of RAM.

Permanent kernel mappings

Permanent kernel mappings allow the kernel to establish long-lasting mappings of
high-memory page frames into the kernel address space. They use a dedicated Page
Table in the master kernel page tables. The `pkmap_page_table` variable stores the
address of this Page Table, while the `LAST_PKMAP` macro yields the number of entries.
As usual, the Page Table includes either 512 or 1,024 entries, according to whether
PAE is enabled or disabled (see the section "The Physical Address Extension (PAE)
Paging Mechanism" in Chapter 2); thus, the kernel can access at most 2 or 4 MB of
high memory at once.

The Page Table maps the linear addresses starting from `PKMAP_BASE`. The `pkmap_count`
array includes `LAST_PKMAP` counters, one for each entry of the `pkmap_page_table` Page
Table. We distinguish three cases:

The counter is 0

> The corresponding Page Table entry does not map any high-memory page frame
> and is usable.

The counter is 1

> The corresponding Page Table entry does not map any high-memory page frame,
> but it cannot be used because the corresponding TLB entry has not been flushed
> since its last usage.

The counter is n (greater than 1)

> The corresponding Page Table entry maps a high-memory page frame, which is
> used by exactly $n-1$ kernel components.

To keep track of the association between high memory page frames and linear
addresses induced by permanent kernel mappings, the kernel makes use of the `page_
address_htable` hash table. This table contains one `page_address_map` data structure
for each page frame in high memory that is currently mapped. In turn, this data
structure contains a pointer to the page descriptor and the linear address assigned to
the page frame.

The `page_address()` function returns the linear address associated with the page
frame, or `NULL` if the page frame is in high memory and is not mapped. This func-
tion, which receives as its parameter a page descriptor pointer `page`, distinguishes
two cases:

1. If the page frame is not in high memory (`PG_highmem` flag clear), the linear address
   always exists and is obtained by computing the page frame index, converting it
   into a physical address, and finally deriving the linear address corresponding to
   the physical address. This is accomplished by the following code:

   ```
   __va((unsigned long)(page - mem_map) << 12)
   ```

2. If the page frame is in high memory (PG_highmem flag set), the function looks into the page_address_htable hash table. If the page frame is found in the hash table, page_address() returns its linear address, otherwise it returns NULL.

The kmap() function establishes a permanent kernel mapping. It is essentially equivalent to the following code:

```
void * kmap(struct page * page)
{
    if (!PageHighMem(page))
        return page_address(page);
    return kmap_high(page);
}
```

The kmap_high() function is invoked if the page frame really belongs to high memory. The function is essentially equivalent to the following code:

```
void * kmap_high(struct page * page)
{
    unsigned long vaddr;
    spin_lock(&kmap_lock);
    vaddr = (unsigned long) page_address(page);
    if (!vaddr)
        vaddr = map_new_virtual(page);
    pkmap_count[(vaddr-PKMAP_BASE) >> PAGE_SHIFT]++;
    spin_unlock(&kmap_lock);
    return (void *) vaddr;
}
```

The function gets the kmap_lock spin lock to protect the Page Table against concurrent accesses in multiprocessor systems. Notice that there is no need to disable the interrupts, because kmap() cannot be invoked by interrupt handlers and deferrable functions. Next, the kmap_high() function checks whether the page frame is already mapped by invoking page_address(). If not, the function invokes map_new_virtual() to insert the page frame physical address into an entry of pkmap_page_table and to add an element to the page_address_htable hash table. Then kmap_high() increases the counter corresponding to the linear address of the page frame to take into account the new kernel component that invoked this function. Finally, kmap_high() releases the kmap_lock spin lock and returns the linear address that maps the page frame.

The map_new_virtual() function essentially executes two nested loops:

```
for (;;) {
    int count;
    DECLARE_WAITQUEUE(wait, current);
    for (count = LAST_PKMAP; count > 0; --count) {
        last_pkmap_nr = (last_pkmap_nr + 1) & (LAST_PKMAP - 1);
        if (!last_pkmap_nr) {
            flush_all_zero_pkmaps();
            count = LAST_PKMAP;
        }
        if (!pkmap_count[last_pkmap_nr]) {
```

```
                unsigned long vaddr = PKMAP_BASE +
                              (last_pkmap_nr << PAGE_SHIFT);
                set_pte(&(pkmap_page_table[last_pkmap_nr]),
                      mk_pte(page, __pgprot(0x63)));
                pkmap_count[last_pkmap_nr] = 1;
                set_page_address(page, (void *) vaddr);
                return vaddr;
            }
        }
        current->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue(&pkmap_map_wait, &wait);
        spin_unlock(&kmap_lock);
        schedule();
        remove_wait_queue(&pkmap_map_wait, &wait);
        spin_lock(&kmap_lock);
        if (page_address(page))
            return (unsigned long) page_address(page);
    }
```

In the inner loop, the function scans all counters in pkmap_count until it finds a null value. The large if block runs when an unused entry is found in pkmap_count. That block determines the linear address corresponding to the entry, creates an entry for it in the pkmap_page_table Page Table, sets the count to 1 because the entry is now used, invokes set_page_address() to insert a new element in the page_address_htable hash table, and returns the linear address.

The function starts where it left off last time, cycling through the pkmap_count array. It does this by preserving in a variable named last_pkmap_nr the index of the last used entry in the pkmap_page_table Page Table. Thus, the search starts from where it was left in the last invocation of the map_new_virtual() function.

When the last counter in pkmap_count is reached, the search restarts from the counter at index 0. Before continuing, however, map_new_virtual() invokes the flush_all_zero_pkmaps() function, which starts another scan of the counters, looking for those that have the value 1. Each counter that has a value of 1 denotes an entry in pkmap_page_table that is free but cannot be used because the corresponding TLB entry has not yet been flushed. flush_all_zero_pkmaps() resets their counters to zero, deletes the corresponding elements from the page_address_htable hash table, and issues TLB flushes on all entries of pkmap_page_table.

If the inner loop cannot find a null counter in pkmap_count, the map_new_virtual() function blocks the current process until some other process releases an entry of the pkmap_page_table Page Table. This is achieved by inserting current in the pkmap_map_wait wait queue, setting the current state to TASK_UNINTERRUPTIBLE, and invoking schedule() to relinquish the CPU. Once the process is awakened, the function checks whether another process has mapped the page by invoking page_address(); if no other process has mapped the page yet, the inner loop is restarted.

The `kunmap()` function destroys a permanent kernel mapping established previously by `kmap()`. If the page is really in the high memory zone, it invokes the `kunmap_high()` function, which is essentially equivalent to the following code:

```
void kunmap_high(struct page * page)
{
    spin_lock(&kmap_lock);
    if ((--pkmap_count[((unsigned long)page_address(page)
                        -PKMAP_BASE)>>PAGE_SHIFT]) == 1)
        if (waitqueue_active(&pkmap_map_wait))
            wake_up(&pkmap_map_wait);
    spin_unlock(&kmap_lock);
}
```

The expression within the brackets computes the index into the `pkmap_count` array from the page's linear address. The counter is decreased and compared to 1. A successful comparison indicates that no process is using the page. The function can finally wake up processes in the wait queue filled by `map_new_virtual()`, if any.

Temporary kernel mappings

Temporary kernel mappings are simpler to implement than permanent kernel mappings; moreover, they can be used inside interrupt handlers and deferrable functions, because requesting a temporary kernel mapping never blocks the current process.

Every page frame in high memory can be mapped through a *window* in the kernel address space—namely, a Page Table entry that is reserved for this purpose. The number of windows reserved for temporary kernel mappings is quite small.

Each CPU has its own set of 13 windows, represented by the enum `km_type` data structure. Each symbol defined in this data structure—such as `KM_BOUNCE_READ`, `KM_USER0`, or `KM_PTE0`—identifies the linear address of a window.

The kernel must ensure that the same window is never used by two kernel control paths at the same time. Thus, each symbol in the `km_type` structure is dedicated to one kernel component and is named after the component. The last symbol, `KM_TYPE_NR`, does not represent a linear address by itself, but yields the number of different windows usable by every CPU.

Each symbol in `km_type`, except the last one, is an index of a fix-mapped linear address (see the section "Fix-Mapped Linear Addresses" in Chapter 2). The enum `fixed_addresses` data structure includes the symbols `FIX_KMAP_BEGIN` and `FIX_KMAP_END`; the latter is assigned to the index `FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1`. In this manner, there are `KM_TYPE_NR` fix-mapped linear addresses for each CPU in the system. Furthermore, the kernel initializes the `kmap_pte` variable with the address of the Page Table entry corresponding to the `fix_to_virt(FIX_KMAP_BEGIN)` linear address.

To establish a temporary kernel mapping, the kernel invokes the `kmap_atomic()` function, which is essentially equivalent to the following code:

```
void * kmap_atomic(struct page * page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;

    current_thread_info()->preempt_count++;
    if (!PageHighMem(page))
        return page_address(page);
    idx = type + KM_TYPE_NR * smp_processor_id();
    vaddr = fix_to_virt(FIX_KMAP_BEGIN + idx);
    set_pte(kmap_pte-idx, mk_pte(page, 0x063));
    __flush_tlb_single(vaddr);
    return (void *) vaddr;
}
```

The `type` argument and the CPU identifier retrieved through `smp_processor_id()` specify what fix-mapped linear address has to be used to map the request page. The function returns the linear address of the page frame if it doesn't belong to high memory; otherwise, it sets up the Page Table entry corresponding to the fix-mapped linear address with the page's physical address and the bits `Present`, `Accessed`, `Read/Write`, and `Dirty`. Finally, the function flushes the proper TLB entry and returns the linear address.

To destroy a temporary kernel mapping, the kernel uses the `kunmap_atomic()` function. In the 80×86 architecture, this function decreases the `preempt_count` of the current process; thus, if the kernel control path was preemptable right before requiring a temporary kernel mapping, it will be preemptable again after it has destroyed the same mapping. Moreover, `kunmap_atomic()` checks whether the `TIF_NEED_RESCHED` flag of `current` is set and, if so, invokes `schedule()`.

The Buddy System Algorithm

The kernel must establish a robust and efficient strategy for allocating groups of contiguous page frames. In doing so, it must deal with a well-known memory management problem called *external fragmentation*: frequent requests and releases of groups of contiguous page frames of different sizes may lead to a situation in which several small blocks of free page frames are "scattered" inside blocks of allocated page frames. As a result, it may become impossible to allocate a large block of contiguous page frames, even if there are enough free pages to satisfy the request.

There are essentially two ways to avoid external fragmentation:

- Use the paging circuitry to map groups of noncontiguous free page frames into intervals of contiguous linear addresses.

- Develop a suitable technique to keep track of the existing blocks of free contiguous page frames, avoiding as much as possible the need to split up a large free block to satisfy a request for a smaller one.

The second approach is preferred by the kernel for three good reasons:

- In some cases, contiguous page frames are really necessary, because contiguous linear addresses are not sufficient to satisfy the request. A typical example is a memory request for buffers to be assigned to a DMA processor (see Chapter 13). Because most DMAs ignore the paging circuitry and access the address bus directly while transferring several disk sectors in a single I/O operation, the buffers requested must be located in contiguous page frames.

- Even if contiguous page frame allocation is not strictly necessary, it offers the big advantage of leaving the kernel paging tables unchanged. What's wrong with modifying the Page Tables? As we know from Chapter 2, frequent Page Table modifications lead to higher average memory access times, because they make the CPU flush the contents of the translation lookaside buffers.

- Large chunks of contiguous physical memory can be accessed by the kernel through 4 MB pages. This reduces the translation lookaside buffers misses, thus significantly speeding up the average memory access time (see the section "Translation Lookaside Buffers (TLB)" in Chapter 2).

The technique adopted by Linux to solve the external fragmentation problem is based on the well-known *buddy system* algorithm. All free page frames are grouped into 11 lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 contiguous page frames, respectively. The largest request of 1024 page frames corresponds to a chunk of 4 MB of contiguous RAM. The physical address of the first page frame of a block is a multiple of the group size—for example, the initial address of a 16-page-frame block is a multiple of 16×2^{12} ($2^{12} = 4{,}096$, which is the regular page size).

We'll show how the algorithm works through a simple example:

Assume there is a request for a group of 256 contiguous page frames (i.e., one megabyte). The algorithm checks first to see whether a free block in the 256-page-frame list exists. If there is no such block, the algorithm looks for the next larger block—a free block in the 512-page-frame list. If such a block exists, the kernel allocates 256 of the 512 page frames to satisfy the request and inserts the remaining 256 page frames into the list of free 256-page-frame blocks. If there is no free 512-page block, the kernel then looks for the next larger block (i.e., a free 1024-page-frame block). If such a block exists, it allocates 256 of the 1024 page frames to satisfy the request, inserts the first 512 of the remaining 768 page frames into the list of free 512-page-frame blocks, and inserts the last 256 page frames into the list of free 256-page-frame blocks. If the list of 1024-page-frame blocks is empty, the algorithm gives up and signals an error condition.

The reverse operation, releasing blocks of page frames, gives rise to the name of this algorithm. The kernel attempts to merge pairs of free buddy blocks of size b together into a single block of size $2b$. Two blocks are considered buddies if:

- Both blocks have the same size, say b.
- They are located in contiguous physical addresses.
- The physical address of the first page frame of the first block is a multiple of $2 \times b \times 2^{12}$.

The algorithm is iterative; if it succeeds in merging released blocks, it doubles b and tries again so as to create even bigger blocks.

Data structures

Linux 2.6 uses a different buddy system for each zone. Thus, in the 80×86 architecture, there are 3 buddy systems: the first handles the page frames suitable for ISA DMA, the second handles the "normal" page frames, and the third handles the high-memory page frames. Each buddy system relies on the following main data structures:

- The mem_map array introduced previously. Actually, each zone is concerned with a subset of the mem_map elements. The first element in the subset and its number of elements are specified, respectively, by the zone_mem_map and size fields of the zone descriptor.
- An array consisting of eleven elements of type free_area, one element for each group size. The array is stored in the free_area field of the zone descriptor.

Let us consider the kth element of the free_area array in the zone descriptor, which identifies all the free blocks of size 2^k. The free_list field of this element is the head of a doubly linked circular list that collects the page descriptors associated with the free blocks of 2^k pages. More precisely, this list includes the page descriptors of the starting page frame of every block of 2^k free page frames; the pointers to the adjacent elements in the list are stored in the lru field of the page descriptor.[*]

Besides the head of the list, the kth element of the free_area array includes also the field nr_free, which specifies the number of free blocks of size 2^k pages. Of course, if there are no blocks of 2^k free page frames, nr_free is equal to 0 and the free_list list is empty (both pointers of free_list point to the free_list field itself).

Finally, the private field of the descriptor of the first page in a block of 2^k free pages stores the order of the block, that is, the number k. Thanks to this field, when a block of pages is freed, the kernel can determine whether the buddy of the block is also free and, if so, it can coalesce the two blocks in a single block of 2^{k+1} pages. It

---

[*] As we'll see later, the lru field of the page descriptor can be used with other meanings when the page is not free.

should be noted that up to Linux 2.6.10, the kernel used 10 arrays of flags to encode this information.

Allocating a block

The `__rmqueue()` function is used to find a free block in a zone. The function takes two arguments: the address of the zone descriptor, and `order`, which denotes the logarithm of the size of the requested block of free pages (0 for a one-page block, 1 for a two-page block, and so forth). If the page frames are successfully allocated, the `__rmqueue()` function returns the address of the page descriptor of the first allocated page frame. Otherwise, the function returns `NULL`.

The `__rmqueue()` function assumes that the caller has already disabled local interrupts and acquired the `zone->lock` spin lock, which protects the data structures of the buddy system. It performs a cyclic search through each list for an available block (denoted by an entry that doesn't point to the entry itself), starting with the list for the requested `order` and continuing if necessary to larger orders:

```
struct free_area *area;
unsigned int current_order;

for (current_order=order; current_order<11; ++current_order) {
    area = zone->free_area + current_order;
    if (!list_empty(&area->free_list))
        goto block_found;
}
return NULL;
```

If the loop terminates, no suitable free block has been found, so `__rmqueue()` returns a `NULL` value. Otherwise, a suitable free block has been found; in this case, the descriptor of its first page frame is removed from the list and the value of `free_pages` in the zone descriptor is decreased:

```
block_found:
    page = list_entry(area->free_list.next, struct page, lru);
    list_del(&page->lru);
    ClearPagePrivate(page);
    page->private = 0;
    area->nr_free--;
    zone->free_pages -= 1UL << order;
```

If the block found comes from a list of size `curr_order` greater than the requested size `order`, a `while` cycle is executed. The rationale behind these lines of codes is as follows: when it becomes necessary to use a block of 2^k page frames to satisfy a request for 2^h page frames ($h < k$), the program allocates the first 2^h page frames and iteratively reassigns the last $2^k - 2^h$ page frames to the `free_area` lists that have indexes between h and k:

```
size = 1 << curr_order;
while (curr_order > order) {
    area--;
    curr_order--;
```

```
        size >>= 1;
        buddy = page + size;
        /* insert buddy as first element in the list */
        list_add(&buddy->lru, &area->free_list);
        area->nr_free++;
        buddy->private = curr_order;
        SetPagePrivate(buddy);
    }
    return page;
```

Because the `__rmqueue()` function has found a suitable free block, it returns the address page of the page descriptor associated with the first allocated page frame.

Freeing a block

The `__free_pages_bulk()` function implements the buddy system strategy for freeing page frames. It uses three basic input parameters:[*]

page
> The address of the descriptor of the first page frame included in the block to be released

zone
> The address of the zone descriptor

order
> The logarithmic size of the block

The function assumes that the caller has already disabled local interrupts and acquired the `zone->lock` spin lock, which protects the data structure of the buddy system. `__free_pages_bulk()` starts by declaring and initializing a few local variables:

```
struct page * base = zone->zone_mem_map;
unsigned long buddy_idx, page_idx = page - base;
struct page * buddy, * coalesced;
int order_size = 1 << order;
```

The `page_idx` local variable contains the index of the first page frame in the block with respect to the first page frame of the zone.

The `order_size` local variable is used to increase the counter of free page frames in the zone:

```
zone->free_pages += order_size;
```

The function now performs a cycle executed at most 10−order times, once for each possibility for merging a block with its buddy. The function starts with the smallest-sized block and moves up to the top size:

```
while (order < 10) {
    buddy_idx = page_idx ^ (1 << order);
```

---

[*] For performance reasons, this inline function also uses another parameter; its value, however, can be determined by the three basic parameters shown in the text.

```
        buddy = base + buddy_idx;
        if (!page_is_buddy(buddy, order))
            break;
        list_del(&buddy->lru);
        zone->free_area[order].nr_free--;
        ClearPagePrivate(buddy);
        buddy->private = 0;
        page_idx &= buddy_idx;
        order++;
    }
```

In the body of the loop, the function looks for the index `buddy_idx` of the block, which is buddy to the one having the page descriptor index `page_idx`. It turns out that this index can be easily computed as:

```
buddy_idx = page_idx ^ (1 << order);
```

In fact, an Exclusive OR (XOR) using the `(1<<order)` mask switches the value of the order-th bit of `page_idx`. Therefore, if the bit was previously zero, `buddy_idx` is equal to `page_idx` + `order_size`; conversely, if the bit was previously one, `buddy_idx` is equal to `page_idx` – `order_size`.

Once the buddy block index is known, the page descriptor of the buddy block can be easily obtained as:

```
buddy = base + buddy_idx;
```

Now the function invokes `page_is_buddy()` to check if `buddy` describes the first page of a block of `order_size` free page frames.

```
int page_is_buddy(struct page *page, int order)
{
    if (PagePrivate(buddy) && page->private == order &&
            !PageReserved(buddy) && page_count(page) ==0)
        return 1;
    return 0;
}
```

As you see, the buddy's first page must be free (`_count` field equal to -1), it must belong to the dynamic memory (`PG_reserved` bit clear), its `private` field must be meaningful (`PG_private` bit set), and finally the `private` field must store the order of the block being freed.

If all these conditions are met, the buddy block is free and the function removes the buddy block from the list of free blocks of order `order`, and performs one more iteration looking for buddy blocks twice as big.

If at least one of the conditions in `page_is_buddy()` is not met, the function breaks out of the cycle, because the free block obtained cannot be merged further with other free blocks. The function inserts it in the proper list and updates the `private` field of the first page frame with the order of the block size:

```
coalesced = base + page_idx;
coalesced->private = order;
```

```
SetPagePrivate(coalesced);
list_add(&coalesced->lru, &zone->free_area[order].free_list);
zone->free_area[order].nr_free++;
```

The Per-CPU Page Frame Cache

As we will see later in this chapter, the kernel often requests and releases single page frames. To boost system performance, each memory zone defines a *per-CPU page frame cache*. Each per-CPU cache includes some pre-allocated page frames to be used for single memory requests issued by the local CPU.

Actually, there are two caches for each memory zone and for each CPU: a *hot cache*, which stores page frames whose contents are likely to be included in the CPU's hardware cache, and a *cold cache*.

Taking a page frame from the hot cache is beneficial for system performance if either the kernel or a User Mode process will write into the page frame right after the allocation. In fact, every access to a memory cell of the page frame will result in a line of the hardware cache being "stolen" from another page frame—unless, of course, the hardware cache already includes a line that maps the cell of the "hot" page frame just accessed.

Conversely, taking a page frame from the cold cache is convenient if the page frame is going to be filled with a DMA operation. In this case, the CPU is not involved and no line of the hardware cache will be modified. Taking the page frame from the cold cache preserves the reserve of hot page frames for the other kinds of memory allocation requests.

The main data structure implementing the per-CPU page frame cache is an array of `per_cpu_pageset` data structures stored in the `pageset` field of the memory zone descriptor. The array includes one element for each CPU; this element, in turn, consists of two `per_cpu_pages` descriptors, one for the hot cache and the other for the cold cache. The fields of the `per_cpu_pages` descriptor are listed in Table 8-7.

Table 8-7. The fields of the per_cpu_pages descriptor

Type	Name	Description
int	count	Number of pages frame in the cache
int	low	Low watermark for cache replenishing
int	high	High watermark for cache depletion
int	batch	Number of page frames to be added or subtracted from the cache
struct list_head	list	List of descriptors of the page frames included in the cache

The kernel monitors the size of the both the hot and cold caches by using two watermarks: if the number of page frames falls below the `low` watermark, the kernel replenishes the proper cache by allocating `batch` single page frames from the buddy

system; otherwise, if the number of page frames rises above the `high` watermark, the kernel releases to the buddy system `batch` page frames in the cache. The values of `batch`, `low`, and `high` essentially depend on the number of page frames included in the memory zone.

Allocating page frames through the per-CPU page frame caches

The `buffered_rmqueue()` function allocates page frames in a given memory zone. It makes use of the per-CPU page frame caches to handle single page frame requests.

The parameters are the address of the memory zone descriptor, the order of the memory allocation request `order`, and the allocation flags `gfp_flags`. If the `__GFP_COLD` flag is set in `gfp_flags`, the page frame should be taken from the cold cache, otherwise it should be taken from the hot cache (this flag is meaningful only for single page frame requests). The function essentially executes the following operations:

1. If `order` is not equal to 0, the per-CPU page frame cache cannot be used: the function jumps to step 4.

2. Checks whether the memory zone's local per-CPU cache identified by the value of the `__GFP_COLD` flag has to be replenished (the `count` field of the `per_cpu_pages` descriptor is lower than or equal to the `low` field). In this case, it executes the following substeps:

    a. Allocates `batch` single page frames from the buddy system by repeatedly invoking the `__rmqueue()` function.

    b. Inserts the descriptors of the allocated page frames in the cache's list.

    c. Updates the value of `count` by adding the number of page frames actually allocated.

3. If `count` is positive, the function gets a page frame from the cache's list, decreases `count`, and jumps to step 5. (Observe that a per-CPU page frame cache could be empty; this happens when the `__rmqueue()` function invoked in step 2a fails to allocate any page frames.)

4. Here, the memory request has not yet been satisfied, either because the request spans several contiguous page frames, or because the selected page frame cache is empty. Invokes the `__rmqueue()` function to allocate the requested page frames from the buddy system.

5. If the memory request has been satisfied, the function initializes the page descriptor of the (first) page frame: clears some flags, sets the `private` field to zero, and sets the page frame reference counter to one. Moreover, if the `__GPF_ZERO` flag in `gfp_flags` is set, it fills the allocated memory area with zeros.

6. Returns the page descriptor address of the (first) page frame, or `NULL` if the memory allocation request failed.

Releasing page frames to the per-CPU page frame caches

In order to release a single page frame to a per-CPU page frame cache, the kernel makes use of the `free_hot_page()` and `free_cold_page()` functions. Both of them are simple wrappers for the `free_hot_cold_page()` function, which receives as its parameters the descriptor address page of the page frame to be released and a `cold` flag specifying either the hot cache or the cold cache.

The `free_hot_cold_page()` function executes the following operations:

1. Gets from the `page->flags` field the address of the memory zone descriptor including the page frame (see the earlier section "Non-Uniform Memory Access (NUMA)").

2. Gets the address of the `per_cpu_pages` descriptor of the zone's cache selected by the `cold` flag.

3. Checks whether the cache should be depleted: if `count` is higher than or equal to `high`, invokes the `free_pages_bulk()` function, passing to it the zone descriptor, the number of page frames to be released (`batch` field), the address of the cache's list, and the number zero (for 0-order page frames). In turn, the latter function invokes repeatedly the `__free_pages_bulk()` function to releases the specified number of page frames—taken from the cache's list—to the buddy system of the memory zone.

4. Adds the page frame to be released to the cache's list, and increases the `count` field.

It should be noted that in the current version of the Linux 2.6 kernel, no page frame is ever released to the cold cache: the kernel always assumes the freed page frame is hot with respect to the hardware cache. Of course, this does not mean that the cold cache is empty: the cache is replenished by `buffered_rmqueue()` when the low watermark has been reached.

The Zone Allocator

The *zone allocator* is the frontend of the kernel page frame allocator. This component must locate a memory zone that includes a number of free page frames large enough to satisfy the memory request. This task is not as simple as it could appear at a first glance, because the zone allocator must satisfy several goals:

- It should protect the pool of reserved page frames (see the earlier section "The Pool of Reserved Page Frames").

- It should trigger the page frame reclaiming algorithm (see Chapter 17) when memory is scarce and blocking the current process is allowed; once some page frames have been freed, the zone allocator will retry the allocation.

- It should preserve the small, precious `ZONE_DMA` memory zone, if possible. For instance, the zone allocator should be somewhat reluctant to assign page frames

in the `ZONE_DMA` memory zone if the request was for `ZONE_NORMAL` or `ZONE_HIGHMEM` page frames.

We have seen in the earlier section "The Zoned Page Frame Allocator" that every request for a group of contiguous page frames is eventually handled by executing the `alloc_pages` macro. This macro, in turn, ends up invoking the `__alloc_pages()` function, which is the core of the zone allocator. It receives three parameters:

`gfp_mask`
> The flags specified in the memory allocation request (see earlier Table 8-5)

`order`
> The logarithmic size of the group of contiguous page frames to be allocated

`zonelist`
> Pointer to a `zonelist` data structure describing, in order of preference, the memory zones suitable for the memory allocation

The `__alloc_pages()` function scans every memory zone included in the `zonelist` data structure. The code that does this looks like the following:

```
for (i = 0; (z=zonelist->zones[i]) != NULL; i++) {
    if (zone_watermark_ok(z, order, ...)) {
        page = buffered_rmqueue(z, order, gfp_mask);
        if (page)
            return page;
    }
}
```

For each memory zone, the function compares the number of free page frames with a threshold value that depends on the memory allocation flags, on the type of current process, and on how many times the zone has already been checked by the function. In fact, if free memory is scarce, every memory zone is typically scanned several times, each time with lower threshold on the minimal amount of free memory required for the allocation. The previous block of code is thus replicated several times—with minor variations—in the body of the `__alloc_pages()` function. The `buffered_rmqueue()` function has been described already in the earlier section "The Per-CPU Page Frame Cache:" it returns the page descriptor of the first allocated page frame, or `NULL` if the memory zone does not include a group of contiguous page frames of the requested size.

The `zone_watermark_ok()` auxiliary function receives several parameters, which determine a threshold `min` on the number of free page frames in the memory zone. In particular, the function returns the value 1 if the following two conditions are met:

1. Besides the page frames to be allocated, there are at least `min` free page frames in the memory zone, not including the page frames in the low-on-memory reserve (`lowmem_reserve` field of the zone descriptor).

2. Besides the page frames to be allocated, there are at least $min/2^k$ free page frames in blocks of order at least k, for each k between 1 and the order of the

allocation. Therefore, if `order` is greater than zero, there must be at least `min/2` free page frames in blocks of size at least 2; if `order` is greater than one, there must be at least `min/4` free page frames in blocks of size at least 4; and so on.

The value of the threshold `min` is determined by `zone_watermark_ok()` as follows:

- The base value is passed as a parameter of the function and can be one of the `pages_min`, `pages_low`, and `pages_high` zone's watermarks (see the section "The Pool of Reserved Page Frames" earlier in this chapter).

- The base value is divided by two if the `gfp_high` flag passed as parameter is set. Usually, this flag is equal to one if the `__GFP_HIGHMEM` flag is set in the `gfp_mask`, that is, if the page frames can be allocated from high memory.

- The threshold value is further reduced by one-fourth if the `can_try_harder` flag passed as parameter is set. This flag is usually equal to one if either the `__GFP_WAIT` flag is set in `gfp_mask`, or if the current process is a real-time process and the memory allocation is done in process context (outside of interrupt handlers and deferrable functions).

The `__alloc_pages()` function essentially executes the following steps:

1. Performs a first scanning of the memory zones (see the block of code shown earlier). In this first scan, the `min` threshold value is set to `z->pages_low`, where `z` points to the zone descriptor being analyzed (the `can_try_harder` and `gfp_high` parameters are set to zero).

2. If the function did not terminate in the previous step, there is not much free memory left: the function awakens the *kswapd* kernel threads to start reclaiming page frames asynchronously (see Chapter 17).

3. Performs a second scanning of the memory zones, passing as base threshold the value `z->pages_min`. As explained previously, the actual threshold is determined also by the `can_try_harder` and `gfp_high` flags. This step is nearly identical to step 1, except that the function is using a lower threshold.

4. If the function did not terminate in the previous step, the system is definitely low on memory. If the kernel control path that issued the memory allocation request is not an interrupt handler or a deferrable function and it is trying to reclaim page frames (either the `PF_MEMALLOC` flag or the `PF_MEMDIE` flag of `current` is set), the function then performs a third scanning of the memory zones, trying to allocate the page frames ignoring the low-on-memory thresholds—that is, without invoking `zone_watermark_ok()`. This is the only case where the kernel control path is allowed to deplete the low-on-memory reserve of pages specified by the `lowmem_reserve` field of the zone descriptor. In fact, in this case the kernel control path that issued the memory request is ultimately trying to free page frames, thus it should get what it has requested, if at all possible. If no memory zone includes enough page frames, the function returns `NULL` to notify the caller of the failure.

5. Here, the invoking kernel control path is not trying to reclaim memory. If the `__GFP_WAIT` flag of `gfp_mask` is not set, the function returns `NULL` to notify the kernel control path of the memory allocation failure: in this case, there is no way to satisfy the request without blocking the current process.

6. Here the current process can be blocked: invokes `cond_resched()` to check whether some other process needs the CPU.

7. Sets the `PF_MEMALLOC` flag of `current`, to denote the fact that the process is ready to perform memory reclaiming.

8. Stores in `current->reclaim_state` a pointer to a `reclaim_state` structure. This structure includes just one field, `reclaimed_slab`, initialized to zero (we'll see how this field is used in the section "Interfacing the Slab Allocator with the Zoned Page Frame Allocator" later in this chapter).

9. Invokes `try_to_free_pages()` to look for some page frames to be reclaimed (see the section "Low On Memory Reclaiming" in Chapter 17). The latter function may block the current process. Once that function returns, `__alloc_pages()` resets the `PF_MEMALLOC` flag of `current` and invokes once more `cond_resched()`.

10. If the previous step has freed some page frames, the function performs yet another scanning of the memory zones equal to the one performed in step 3. If the memory allocation request cannot be satisfied, the function determines whether it should continue scanning the memory zone: if the `__GFP_NORETRY` flag is clear and either the memory allocation request spans up to eight page frames, or one of the `__GFP_REPEAT` and `__GFP_NOFAIL` flags is set, the function invokes `blk_congestion_wait()` to put the process asleep for awhile (see Chapter 14), and it jumps back to step 6. Otherwise, the function returns `NULL` to notify the caller that the memory allocation failed.

11. If no page frame has been freed in step 9, the kernel is in deep trouble, because free memory is dangerously low and it was not possible to reclaim any page frame. Perhaps the time has come to take a crucial decision. If the kernel control path is allowed to perform the filesystem-dependent operations needed to kill a process (the `__GFP_FS` flag in `gfp_mask` is set) and the `__GFP_NORETRY` flag is clear, performs the following substeps:

    a. Scans once again the memory zones with a threshold value equal to `z->pages_high`.

    b. Invokes `out_of_memory()` to start freeing some memory by killing a victim process (see "The Out of Memory Killer" in Chapter 17).

    c. Jumps back to step 1.

Because the watermark used in step 11a is much higher than the watermarks used in the previous scannings, that step is likely to fail. Actually, step 11a succeeds only if another kernel control path is already killing a process to reclaim its memory. Thus, step 11a avoids that two innocent processes are killed instead of one.

Releasing a group of page frames

The zone allocator also takes care of releasing page frames; thankfully, releasing memory is a lot easier than allocating it.

All kernel macros and functions that release page frames—described in the earlier section "The Zoned Page Frame Allocator"—rely on the __free_pages() function. It receives as its parameters the address of the page descriptor of the first page frame to be released (page), and the logarithmic size of the group of contiguous page frames to be released (order). The function executes the following steps:

1. Checks that the first page frame really belongs to dynamic memory (its PG_ reserved flag is cleared); if not, terminates.

2. Decreases the page->_count usage counter; if it is still greater than or equal to zero, terminates.

3. If order is equal to zero, the function invokes free_hot_page() to release the page frame to the per-CPU hot cache of the proper memory zone (see the earlier section "The Per-CPU Page Frame Cache").

4. If order is greater than zero, it adds the page frames in a local list and invokes the free_pages_bulk() function to release them to the buddy system of the proper memory zone (see step 3 in the description of free_hot_cold_page() in the earlier section "The Per-CPU Page Frame Cache").

Memory Area Management

This section deals with *memory areas*—that is, with sequences of memory cells having contiguous physical addresses and an arbitrary length.

The buddy system algorithm adopts the page frame as the basic memory area. This is fine for dealing with relatively large memory requests, but how are we going to deal with requests for small memory areas, say a few tens or hundreds of bytes?

Clearly, it would be quite wasteful to allocate a full page frame to store a few bytes. A better approach instead consists of introducing new data structures that describe how small memory areas are allocated within the same page frame. In doing so, we introduce a new problem called *internal fragmentation*. It is caused by a mismatch between the size of the memory request and the size of the memory area allocated to satisfy the request.

A classical solution (adopted by early Linux versions) consists of providing memory areas whose sizes are geometrically distributed; in other words, the size depends on a power of 2 rather than on the size of the data to be stored. In this way, no matter what the memory request size is, we can ensure that the internal fragmentation is always smaller than 50 percent. Following this approach, the kernel creates 13 geometrically distributed lists of free memory areas whose sizes range from 32 to 131, 072 bytes. The buddy system is invoked both to obtain additional page frames

needed to store new memory areas and, conversely, to release page frames that no longer contain memory areas. A dynamic list is used to keep track of the free memory areas contained in each page frame.

The Slab Allocator

Running a memory area allocation algorithm on top of the buddy algorithm is not particularly efficient. A better algorithm is derived from the *slab allocator* schema that was adopted for the first time in the Sun Microsystems Solaris 2.4 operating system. It is based on the following premises:

- The type of data to be stored may affect how memory areas are allocated; for instance, when allocating a page frame to a User Mode process, the kernel invokes the get_zeroed_page( ) function, which fills the page with zeros.

  The concept of a slab allocator expands upon this idea and views the memory areas as *objects* consisting of both a set of data structures and a couple of functions or methods called the *constructor* and *destructor*. The former initializes the memory area while the latter deinitializes it.

  To avoid initializing objects repeatedly, the slab allocator does not discard the objects that have been allocated and then released but instead saves them in memory. When a new object is then requested, it can be taken from memory without having to be reinitialized.

- The kernel functions tend to request memory areas of the same type repeatedly. For instance, whenever the kernel creates a new process, it allocates memory areas for some fixed size tables such as the process descriptor, the open file object, and so on (see Chapter 3). When a process terminates, the memory areas used to contain these tables can be reused. Because processes are created and destroyed quite frequently, without the slab allocator, the kernel wastes time allocating and deallocating the page frames containing the same memory areas repeatedly; the slab allocator allows them to be saved in a cache and reused quickly.

- Requests for memory areas can be classified according to their frequency. Requests of a particular size that are expected to occur frequently can be handled most efficiently by creating a set of special-purpose objects that have the right size, thus avoiding internal fragmentation. Meanwhile, sizes that are rarely encountered can be handled through an allocation scheme based on objects in a series of geometrically distributed sizes (such as the power-of-2 sizes used in early Linux versions), even if this approach leads to internal fragmentation.

- There is another subtle bonus in introducing objects whose sizes are not geometrically distributed: the initial addresses of the data structures are less prone to be concentrated on physical addresses whose values are a power of 2. This, in turn, leads to better performance by the processor hardware cache.

- Hardware cache performance creates an additional reason for limiting calls to the buddy system allocator as much as possible. Every call to a buddy system function "dirties" the hardware cache, thus increasing the average memory access time. The impact of a kernel function on the hardware cache is called the function *footprint*; it is defined as the percentage of cache overwritten by the function when it terminates. Clearly, large footprints lead to a slower execution of the code executed right after the kernel function, because the hardware cache is by now filled with useless information.

The slab allocator groups objects into *caches*. Each cache is a "store" of objects of the same type. For instance, when a file is opened, the memory area needed to store the corresponding "open file" object is taken from a slab allocator cache named *filp* (for "file pointer").

The area of main memory that contains a cache is divided into *slabs*; each slab consists of one or more contiguous page frames that contain both allocated and free objects (see Figure 8-3).
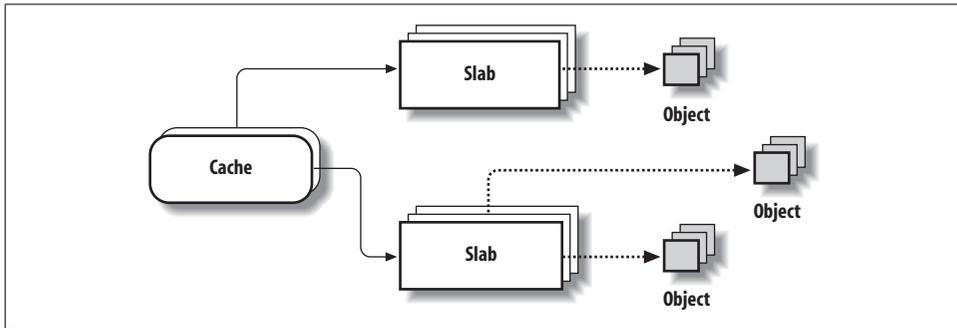
Figure 8-3. The slab allocator components

As we'll see in Chapter 17, the kernel periodically scans the caches and releases the page frames corresponding to empty slabs.

Cache Descriptor

Each cache is described by a structure of type kmem_cache_t (which is equivalent to the type struct kmem_cache_s), whose fields are listed in Table 8-8. We omitted from the table several fields used for collecting statistical information and for debugging.

Table 8-8. The fields of the kmem_cache_t descriptor

Type	Name	Description
struct array_cache * []	array	Per-CPU array of pointers to local caches of free objects (see the section "Local Caches of Free Slab Objects" later in this chapter).
unsigned int	batchcount	Number of objects to be transferred in bulk to or from the local caches.
unsigned int	limit	Maximum number of free objects in the local caches. This is tunable.

Table 8-8. The fields of the kmem_cache_t descriptor (continued)

Type	Name	Description
struct kmem_list3	lists	See next table.
unsigned int	objsize	Size of the objects included in the cache.
unsigned int	flags	Set of flags that describes permanent properties of the cache.
unsigned int	num	Number of objects packed into a single slab. (All slabs of the cache have the same size.)
unsigned int	free_limit	Upper limit of free objects in the whole slab cache.
spinlock_t	spinlock	Cache spin lock.
unsigned int	gfporder	Logarithm of the number of contiguous page frames included in a single slab.
unsigned int	gfpflags	Set of flags passed to the buddy system function when allocating page frames.
size_t	colour	Number of colors for the slabs (see the section "Slab Coloring" later in this chapter).
unsigned int	colour_off	Basic alignment offset in the slabs.
unsigned int	colour_next	Color to use for the next allocated slab.
kmem_cache_t *	slabp_cache	Pointer to the general slab cache containing the slab descriptors (NULL if internal slab descriptors are used; see next section).
unsigned int	slab_size	The size of a single slab.
unsigned int	dflags	Set of flags that describe dynamic properties of the cache.
void *	ctor	Pointer to constructor method associated with the cache.
void *	dtor	Pointer to destructor method associated with the cache.
const char *	name	Character array storing the name of the cache.
struct list_head	next	Pointers for the doubly linked list of cache descriptors.

The lists field of the kmem_cache_t descriptor, in turn, is a structure whose fields are listed in Table 8-9.

Table 8-9. The fields of the kmem_list3 structure

Type	Name	Description
struct list_head	slabs_partial	Doubly linked circular list of slab descriptors with both free and non-free objects
struct list_head	slabs_full	Doubly linked circular list of slab descriptors with no free objects
struct list_head	slabs_free	Doubly linked circular list of slab descriptors with free objects only
unsigned long	free_objects	Number of free objects in the cache
int	free_touched	Used by the slab allocator's page reclaiming algorithm (see Chapter 17)
unsigned long	next_reap	Used by the slab allocator's page reclaiming algorithm (see Chapter 17)

Table 8-9. The fields of the kmem_list3 structure (continued)

Type	Name	Description
struct array_cache *	shared	Pointer to a local cache shared by all CPUs (see the later section "Local Caches of Free Slab Objects")

Slab Descriptor

Each slab of a cache has its own descriptor of type `slab` illustrated in Table 8-10.

Table 8-10. The fields of the slab descriptor

Type	Name	Description
struct list_head	list	Pointers for one of the three doubly linked list of slab descriptors (either the slabs_full, slabs_partial, or slabs_free list in the kmem_list3 structure of the cache descriptor)
unsigned long	colouroff	Offset of the first object in the slab (see the section "Slab Coloring" later in this chapter)
void *	s_mem	Address of first object (either allocated or free) in the slab
unsigned int	inuse	Number of objects in the slab that are currently used (not free)
unsigned int	free	Index of next free object in the slab, or BUFCTL_END if there are no free objects left (see the section "Object Descriptor" later in this chapter)

Slab descriptors can be stored in two possible places:

External slab descriptor

> Stored outside the slab, in one of the general caches not suitable for ISA DMA pointed to by cache_sizes (see the next section).

Internal slab descriptor

> Stored inside the slab, at the beginning of the first page frame assigned to the slab.

The slab allocator chooses the second solution when the size of the objects is smaller than 512MB or when internal fragmentation leaves enough space for the slab descriptor and the object descriptors (as described later)—inside the slab. The CFLGS_ OFF_SLAB flag in the flags field of the cache descriptor is set to one if the slab descriptor is stored outside the slab; it is set to zero otherwise.

Figure 8-4 illustrates the major relationships between cache and slab descriptors. Full slabs, partially full slabs, and free slabs are linked in different lists.

General and Specific Caches

Caches are divided into two types: general and specific. *General caches* are used only by the slab allocator for its own purposes, while *specific caches* are used by the remaining parts of the kernel.
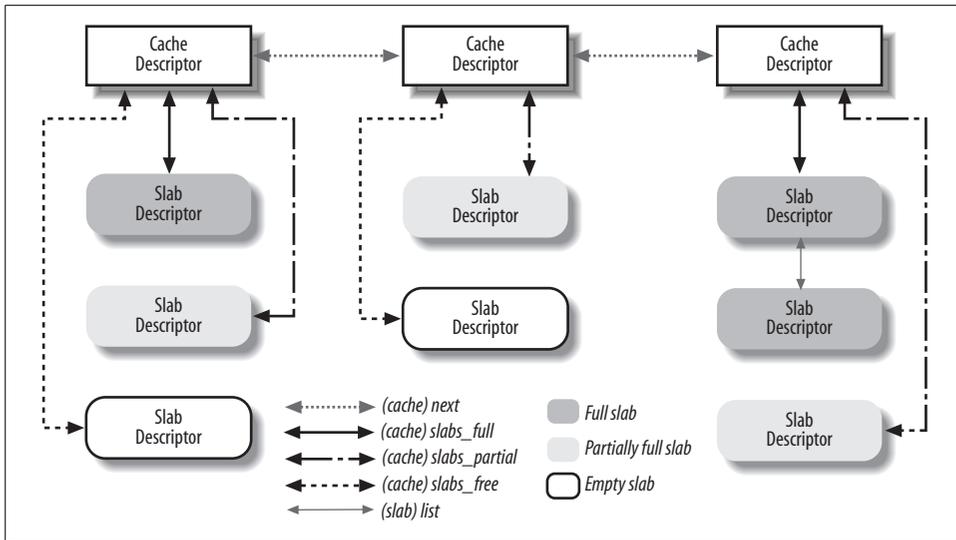
Figure 8-4. Relationship between cache and slab descriptors

The general caches are:

- A first cache called *kmem_cache* whose objects are the cache descriptors of the remaining caches used by the kernel. The cache_cache variable contains the descriptor of this special cache.

- Several additional caches contain general purpose memory areas. The range of the memory area sizes typically includes 13 geometrically distributed sizes. A table called malloc_sizes (whose elements are of type cache_sizes) points to 26 cache descriptors associated with memory areas of size 32, 64, 128, 256, 512, 1,024, 2,048, 4,096, 8,192, 16,384, 32,768, 65,536, and 131,072 bytes. For each size, there are two caches: one suitable for ISA DMA allocations and the other for normal allocations.

The kmem_cache_init( ) function is invoked during system initialization to set up the general caches.

Specific caches are created by the kmem_cache_create( ) function. Depending on the parameters, the function first determines the best way to handle the new cache (for instance, whether to include the slab descriptor inside or outside of the slab). It then allocates a cache descriptor for the new cache from the cache_cache general cache and inserts the descriptor in the cache_chain list of cache descriptors (the insertion is done after having acquired the cache_chain_sem semaphore that protects the list from concurrent accesses).

It is also possible to destroy a cache and remove it from the cache_chain list by invoking kmem_cache_destroy(). This function is mostly useful to modules that create their own caches when loaded and destroy them when unloaded. To avoid wasting memory space, the kernel must destroy all slabs before destroying the cache itself. The kmem_cache_shrink() function destroys all the slabs in a cache by invoking slab_destroy() iteratively (see the later section "Releasing a Slab from a Cache").

The names of all general and specific caches can be obtained at runtime by reading */proc/slabinfo*; this file also specifies the number of free objects and the number of allocated objects in each cache.

Interfacing the Slab Allocator with the Zoned Page Frame Allocator

When the slab allocator creates a new slab, it relies on the zoned page frame allocator to obtain a group of free contiguous page frames. For this purpose, it invokes the kmem_getpages() function, which is essentially equivalent, on a UMA system, to the following code fragment:

```
void * kmem_getpages(kmem_cache_t *cachep, int flags)
{
    struct page *page;
    int i;

    flags |= cachep->gfpflags;
    page = alloc_pages(flags, cachep->gfporder);
    if (!page)
        return NULL;
    i = (1 << cache->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        atomic_add(i, &slab_reclaim_pages);
    while (i--)
            SetPageSlab(page++);
    return page_address(page);
}
```

The two parameters have the following meaning:

cachep

Points to the cache descriptor of the cache that needs additional page frames (the number of required page frames is determined by the order in the cachep->gfporder field).

flags

Specifies how the page frame is requested (see the section "The Zoned Page Frame Allocator" earlier in this chapter). This set of flags is combined with the specific cache allocation flags stored in the gfpflags field of the cache descriptor.

The size of the memory allocation request is specified by the gfporder field of the cache descriptor, which encodes the size of a slab in the cache.* If the slab cache has been created with the SLAB_RECLAIM_ACCOUNT flag set, the page frames assigned to the slabs are accounted for as reclaimable pages when the kernel checks whether there is enough memory to satisfy some User Mode requests. The function also sets the PG_slab flag in the page descriptors of the allocated page frames.

In the reverse operation, page frames assigned to a slab can be released (see the section "Releasing a Slab from a Cache" later in this chapter) by invoking the kmem_freepages( ) function:

```
void kmem_freepages(kmem_cache_t *cachep, void *addr)
{
    unsigned long i = (1<<cachep->gfporder);
    struct page *page = virt_to_page(addr);

    if (current->reclaim_state)
        current->reclaim_state->reclaimed_slab += i;
    while (i--)
        ClearPageSlab(page++);
    free_pages((unsigned long) addr, cachep->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        atomic_sub(1<<cachep->gfporder, &slab_reclaim_pages);
}
```

The function releases the page frames, starting from the one having the linear address addr, that had been allocated to the slab of the cache identified by cachep. If the current process is performing memory reclaiming (current->reclaim_state field not NULL), the reclaimed_slab field of the reclaim_state structure is properly increased, so that the pages just freed can be accounted for by the page frame reclaiming algorithm (see the section "Low On Memory Reclaiming" in Chapter 17). Moreover, if the SLAB_RECLAIM_ACCOUNT flag is set (see above), the slab_reclaim_pages variable is properly decreased.

Allocating a Slab to a Cache

A newly created cache does not contain a slab and therefore does not contain any free objects. New slabs are assigned to a cache only when both of the following are true:

- A request has been issued to allocate a new object.
- The cache does not include a free object.

* Notice that it is not possible to allocate page frames from the ZONE_HIGHMEM memory zone, because the kmem_getpages( ) function returns the linear address yielded by the page_address( ) function; as explained in the section "Kernel Mappings of High-Memory Page Frames" earlier in this chapter, this function returns NULL for unmapped high-memory page frames.

Under these circumstances, the slab allocator assigns a new slab to the cache by invoking `cache_grow()`. This function calls `kmem_getpages()` to obtain from the zoned page frame allocator the group of page frames needed to store a single slab; it then calls `alloc_slabmgmt()` to get a new slab descriptor. If the `CFLGS_OFF_SLAB` flag of the cache descriptor is set, the slab descriptor is allocated from the general cache pointed to by the `slabp_cache` field of the cache descriptor; otherwise, the slab descriptor is allocated in the first page frame of the slab.

The kernel must be able to determine, given a page frame, whether it is used by the slab allocator and, if so, to derive quickly the addresses of the corresponding cache and slab descriptors. Therefore, `cache_grow()` scans all page descriptors of the page frames assigned to the new slab, and loads the `next` and `prev` subfields of the `lru` fields in the page descriptors with the addresses of, respectively, the cache descriptor and the slab descriptor. This works correctly because the `lru` field is used by functions of the buddy system only when the page frame is free, while page frames handled by the slab allocator functions have the `PG_slab` flag set and are not free as far as the buddy system is concerned.[*] The opposite question—given a slab in a cache, which are the page frames that implement it?—can be answered by using the `s_mem` field of the slab descriptor and the `gfporder` field (the size of a slab) of the cache descriptor.

Next, `cache_grow()` calls `cache_init_objs()`, which applies the constructor method (if defined) to all the objects contained in the new slab.

Finally, `cache_grow()` calls `list_add_tail()` to add the newly obtained slab descriptor `*slabp` at the end of the fully free slab list of the cache descriptor `*cachep`, and updates the counter of free objects in the cache:

```
list_add_tail(&slabp->list, &cachep->lists->slabs_free);
cachep->lists->free_objects += cachep->num;
```

Releasing a Slab from a Cache

Slabs can be destroyed in two cases:

- There are too many free objects in the slab cache (see the later section "Releasing a Slab from a Cache").

- A timer function invoked periodically determines that there are fully unused slabs that can be released (see Chapter 17).

In both cases, the `slab_destroy()` function is invoked to destroy a slab and release the corresponding page frames to the zoned page frame allocator:

```
void slab_destroy(kmem_cache_t *cachep, slab_t *slabp)
{
    if (cachep->dtor) {
        int i;
```

---

[*] As we'll in Chapter 17, the `lru` field is also used by the page frame reclaiming algorithm.

```
            for (i = 0; i < cachep->num; i++) {
                void* objp = slabp->s_mem+cachep->objsize*i;
                (cachep->dtor)(objp, cachep, 0);
            }
        }
        kmem_freepages(cachep, slabp->s_mem - slabp->colouroff);
        if (cachep->flags & CFLGS_OFF_SLAB)
            kmem_cache_free(cachep->slabp_cache, slabp);
    }
```

The function checks whether the cache has a destructor method for its objects (the dtor field is not NULL), in which case it applies the destructor to all the objects in the slab; the objp local variable keeps track of the currently examined object. Next, it calls kmem_freepages( ), which returns all the contiguous page frames used by the slab to the buddy system. Finally, if the slab descriptor is stored outside of the slab, the function releases it from the cache of slab descriptors.

Actually, the function is slightly more complicated. For example, a slab cache can be created with the SLAB_DESTROY_BY_RCU flag, which means that slabs should be released in a deferred way by registering a callback with the call_rcu( ) function (see the section "Read-Copy Update (RCU)" in Chapter 5). The callback function, in turn, invokes kmem_freepages() and, possibly, the kmem_cache_free(), as in the main case shown above.

Object Descriptor

Each object has a short descriptor of type kmem_bufctl_t. Object descriptors are stored in an array placed right after the corresponding slab descriptor. Thus, like the slab descriptors themselves, the object descriptors of a slab can be stored in two possible ways that are illustrated in Figure 8-5.

External object descriptors
> Stored outside the slab, in the general cache pointed to by the slabp_cache field of the cache descriptor. The size of the memory area, and thus the particular general cache used to store object descriptors, depends on the number of objects stored in the slab (num field of the cache descriptor).

Internal object descriptors
> Stored inside the slab, right before the objects they describe.

The first object descriptor in the array describes the first object in the slab, and so on. An object descriptor is simply an unsigned short integer, which is meaningful only when the object is free. It contains the index of the next free object in the slab, thus implementing a simple list of free objects inside the slab. The object descriptor of the last element in the free object list is marked by the conventional value BUFCTL_END (0xffff).
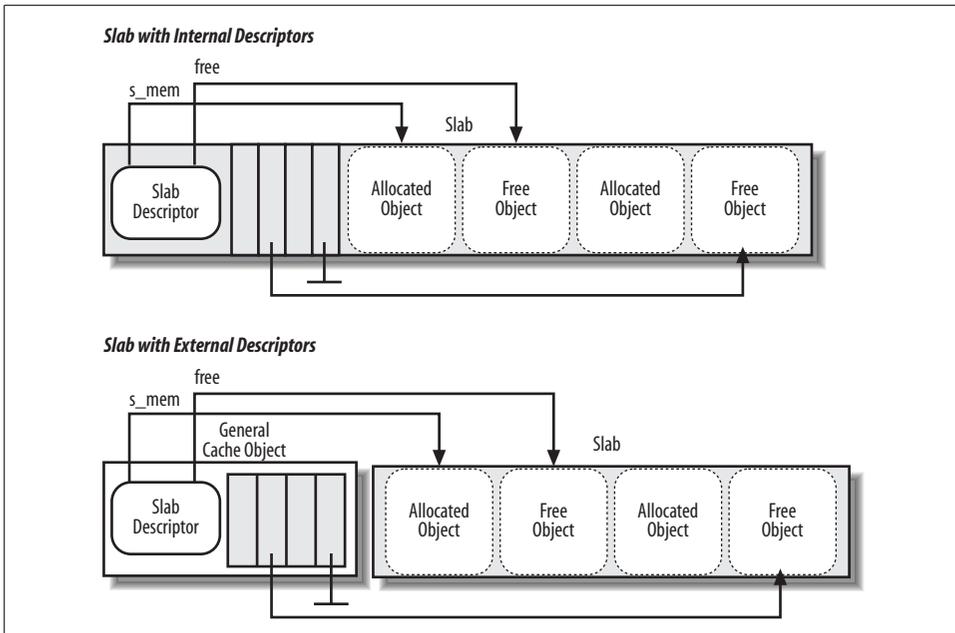
Figure 8-5. Relationships between slab and object descriptors

Aligning Objects in Memory

The objects managed by the slab allocator are *aligned* in memory—that is, they are stored in memory cells whose initial physical addresses are multiples of a given constant, which is usually a power of 2. This constant is called the *alignment factor*.

The largest alignment factor allowed by the slab allocator is 4,096—the page frame size. This means that objects can be aligned by referring to either their physical addresses or their linear addresses. In both cases, only the 12 least significant bits of the address may be altered by the alignment.

Usually, microcomputers access memory cells more quickly if their physical addresses are aligned with respect to the word size (that is, to the width of the internal memory bus of the computer). Thus, by default, the kmem_cache_create( ) function aligns objects according to the word size specified by the BYTES_PER_WORD macro. For 80 × 86 processors, the macro yields the value 4 because the word is 32 bits long.

When creating a new slab cache, it's possible to specify that the objects included in it be aligned in the first-level hardware cache. To achieve this, the kernel sets the SLAB_HWCACHE_ALIGN cache descriptor flag. The kmem_cache_create( ) function handles the request as follows:

- If the object's size is greater than half of a cache line, it is aligned in RAM to a multiple of L1_CACHE_BYTES—that is, at the beginning of the line.

- Otherwise, the object size is rounded up to a submultiple of `L1_CACHE_BYTES`; this ensures that a small object will never span across two cache lines.

Clearly, what the slab allocator is doing here is trading memory space for access time; it gets better cache performance by artificially increasing the object size, thus causing additional internal fragmentation.

Slab Coloring

We know from Chapter 2 that the same hardware cache line maps many different blocks of RAM. In this chapter, we have also seen that objects of the same size end up being stored at the same offset within a cache. Objects that have the same offset within different slabs will, with a relatively high probability, end up mapped in the same cache line. The cache hardware might therefore waste memory cycles transferring two objects from the same cache line back and forth to different RAM locations, while other cache lines go underutilized. The slab allocator tries to reduce this unpleasant cache behavior by a policy called *slab coloring*: different arbitrary values called *colors* are assigned to the slabs.

Before examining slab coloring, we have to look at the layout of objects in the cache. Let's consider a cache whose objects are aligned in RAM. This means that the object address must be a multiple of a given positive value, say *aln*. Even taking the alignment constraint into account, there are many possible ways to place objects inside the slab. The choices depend on decisions made for the following variables:

num
> Number of objects that can be stored in a slab (its value is in the `num` field of the cache descriptor).

osize
> Object size, including the alignment bytes.

dsize
> Slab descriptor size plus all object descriptors size, rounded up to the smallest multiple of the hardware cache line size. Its value is equal to 0 if the slab and object descriptors are stored outside of the slab.

free
> Number of unused bytes (bytes not assigned to any object) inside the slab.

The total length in bytes of a slab can then be expressed as:

$$slab\ length = (num \times osize) + dsize + free$$

free is always smaller than *osize*, because otherwise, it would be possible to place additional objects inside the slab. However, *free* could be greater than *aln*.

The slab allocator takes advantage of the *free* unused bytes to color the slab. The term "color" is used simply to subdivide the slabs and allow the memory allocator to

spread objects out among different linear addresses. In this way, the kernel obtains the best possible performance from the microprocessor's hardware cache.

Slabs having different colors store the first object of the slab in different memory locations, while satisfying the alignment constraint. The number of available colors is *free/aln* (this value is stored in the colour field of the cache descriptor). Thus, the first color is denoted as 0 and the last one is denoted as *(free/aln)*−1. (As a particular case, if *free* is lower than *aln*, colour is set to 0, nevertheless all slabs use color 0, thus really the number of colors is one.)

If a slab is colored with color *col*, the offset of the first object (with respect to the slab initial address) is equal to *col* × *aln* + *dsize* bytes. Figure 8-6 illustrates how the placement of objects inside the slab depends on the slab color. Coloring essentially leads to moving some of the free area of the slab from the end to the beginning.
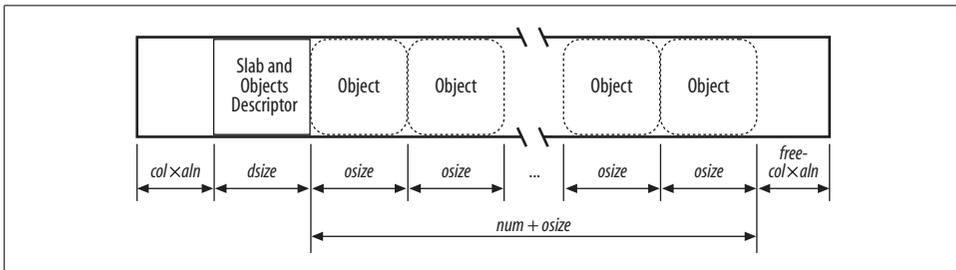
Figure 8-6. Slab with color col and alignment aln

Coloring works only when *free* is large enough. Clearly, if no alignment is required for the objects or if the number of unused bytes inside the slab is smaller than the required alignment (*free* < *aln*), the only possible slab coloring is the one that has the color 0—the one that assigns a zero offset to the first object.

The various colors are distributed equally among slabs of a given object type by storing the current color in a field of the cache descriptor called colour_next. The cache_grow( ) function assigns the color specified by colour_next to a new slab and then increases the value of this field. After reaching colour, it wraps around again to 0. In this way, each slab is created with a different color from the previous one, up to the maximum available colors. The cache_grow( ) function, moreover, gets the value *aln* from the colour_off field of the cache descriptor, computes *dsize* according to the number of objects inside the slab, and finally stores the value *col* × *aln* + *dsize* in the colouroff field of the slab descriptor.

Local Caches of Free Slab Objects

The Linux 2.6 implementation of the slab allocator for multiprocessor systems differs from that of the original Solaris 2.4. To reduce spin lock contention among processors and to make better use of the hardware caches, each cache of the slab

allocator includes a per-CPU data structure consisting of a small array of pointers to freed objects called the *slab local cache*. Most allocations and releases of slab objects affect the local cache only; the slab data structures get involved only when the local cache underflows or overflows. This technique is quite similar to the one illustrated in the section "The Per-CPU Page Frame Cache" earlier in this chapter.

The array field of the cache descriptor is an array of pointers to `array_cache` data structures, one element for each CPU in the system. Each `array_cache` data structure is a descriptor of the local cache of free objects, whose fields are illustrated in Table 8-11.

Table 8-11. The fields of the array_cache structure

Type	Name	Description
unsigned int	avail	Number of pointers to available objects in the local cache. The field also acts as the index of the first free slot in the cache.
unsigned int	limit	Size of the local cache—that is, the maximum number of pointers in the local cache.
unsigned int	batchcount	Chunk size for local cache refill or emptying.
unsigned int	touched	Flag set to 1 if the local cache has been recently used.

Notice that the local cache descriptor does not include the address of the local cache itself; in fact, the local cache is placed right after the descriptor. Of course, the local cache stores the pointers to the freed objects, not the object themselves, which are always placed inside the slabs of the cache.

When creating a new slab cache, the `kmem_cache_create()` function determines the size of the local caches (storing this value in the `limit` field of the cache descriptor), allocates them, and stores their pointers into the `array` field of the cache descriptor. The size depends on the size of the objects stored in the slab cache, and ranges from 1 for very large objects to 120 for small ones. Moreover, the initial value of the `batchcount` field, which is the number of objects added or removed in a chunk from a local cache, is initially set to half of the local cache size.[*]

In multiprocessor systems, slab caches for small objects also sport an additional local cache, whose address is stored in the `lists.shared` field of the cache descriptor. The *shared local cache* is, as the name suggests, shared among all CPUs, and it makes the task of migrating free objects from a local cache to another easier (see the following section). Its initial size is equal to eight times the value of the `batchcount` field.

---

[*] The system administrator can tune—for each cache—the size of the local caches and the value of the batchcount field by writing into the */proc/slabinfo* file.

Allocating a Slab Object

New objects may be obtained by invoking the kmem_cache_alloc() function. The parameter cachep points to the cache descriptor from which the new free object must be obtained, while the parameter flag represents the flags to be passed to the zoned page frame allocator functions, should all slabs of the cache be full.

The function is essentially equivalent to the following:

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
{
    unsigned long save_flags;
    void *objp;
    struct array_cache *ac;

    local_irq_save(save_flags);
    ac = cache_p->array[smp_processor_id()];
    if (ac->avail) {
        ac->touched = 1;
        objp = ((void **)(ac+1))[--ac->avail];
    } else
        objp = cache_alloc_refill(cachep, flags);
    local_irq_restore(save_flags);
    return objp;
}
```

The function tries first to retrieve a free object from the local cache. If there are free objects, the avail field contains the index in the local cache of the entry that points to the last freed object. Because the local cache array is stored right after the ac descriptor, ((void**)(ac+1))[--ac->avail] gets the address of that free object and decreases the value of ac->avail. The cache_alloc_refill() function is invoked to repopulate the local cache and get a free object when there are no free objects in the local cache.

The cache_alloc_refill() function essentially performs the following steps:

1. Stores in the ac local variable the address of the local cache descriptor:

    ac = cachep->array[smp_processor_id()];

2. Gets the cachep->spinlock.

3. If the slab cache includes a shared local cache, and if the shared local cache includes some free objects, it refills the CPU's local cache by moving up to ac-> batchcount pointers from the shared local cache. Then, it jumps to step 6.

4. Tries to fill the local cache with up to ac->batchcount pointers to free objects included in the slabs of the cache:

    a. Looks in the slabs_partial and slabs_free lists of the cache descriptor, and gets the address slabp of a slab descriptor whose corresponding slab is either partially filled or empty. If no such descriptor exists, the function goes to step 5.

b. For each free object in the slab, the function increases the `inuse` field of the slab descriptor, inserts the object's address in the local cache, and updates the `free` field so that it stores the index of the next free object in the slab:

```
slabp->inuse++;
((void**)(ac+1))[ac->avail++] =
                    slabp->s_mem + slabp->free * cachep->obj_size;
slabp->free = ((kmem_bufctl_t*)(slabp+1))[slabp->free];
```

c. Inserts, if necessary, the depleted slab in the proper list, either the `slab_full` or the `slab_partial` list.

5. At this point, the number of pointers added to the local cache is stored in the `ac->avail` field: the function decreases the `free_objects` field of the `kmem_list3` structure of the same amount to specify that the objects are no longer free.

6. Releases the `cachep->spinlock`.

7. If the `ac->avail` field is now greater than 0 (some cache refilling took place), it sets the `ac->touched` field to 1 and returns the free object pointer that was last inserted in the local cache:

```
return ((void**)(ac+1))[--ac->avail];
```

8. Otherwise, no cache refilling took place: invokes `cache_grow()` to get a new slab, and thus new free objects.

9. If `cache_grow()` fails, it returns `NULL`; otherwise it goes back to step 1 to repeat the procedure.

Freeing a Slab Object

The `kmem_cache_free()` function releases an object previously allocated by the slab allocator to some kernel function. Its parameters are `cachep`, the address of the cache descriptor, and `objp`, the address of the object to be released:

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
{
    unsigned long flags;
    struct array_cache *ac;

    local_irq_save(flags);
    ac = cachep->array[smp_processor_id()];
    if (ac->avail == ac->limit)
        cache_flusharray(cachep, ac);
    ((void**)(ac+1))[ac->avail++] = objp;
    local_irq_restore(flags);
}
```

The function checks first whether the local cache has room for an additional pointer to a free object. If so, the pointer is added to the local cache and the function returns. Otherwise it first invokes `cache_flusharray()` to deplete the local cache and then adds the pointer to the local cache.

The `cache_flusharray()` function performs the following operations:

1. Acquires the `cachep->spinlock` spin lock.

2. If the slab cache includes a shared local cache, and if the shared local cache is not already full, it refills the shared local cache by moving up to `ac->batchcount` pointers from the CPU's local cache. Then, it jumps to step 4.

3. Invokes the `free_block()` function to give back to the slab allocator up to `ac->batchcount` objects currently included in the local cache. For each object at address `objp`, the function executes the following steps:

   a. Increases the `lists.free_objects` field of the cache descriptor.

   b. Determines the address of the slab descriptor containing the object:
      ```
      slabp = (struct slab *)(virt_to_page(objp)->lru.prev);
      ```
      (Remember that the `lru.prev` field of the descriptor of the slab page points to the corresponding slab descriptor.)

   c. Removes the slab descriptor from its slab cache list (either `cachep->lists.slabs_partial` or `cachep->lists.slabs_full`).

   d. Computes the index of the object inside the slab:
      ```
      objnr = (objp - slabp->s_mem) / cachep->objsize;
      ```

   e. Stores in the object descriptor the current value of the `slabp->free`, and puts in `slabp->free` the index of the object (the last released object will be the first object to be allocated again):
      ```
      ((kmem_bufctl_t *)(slabp+1))[objnr] = slabp->free;
      slabp->free = objnr;
      ```

   f. Decreases the `slabp->inuse` field.

   g. If `slabp->inuse` is equal to zero—all objects in the slab are free—and the number of free objects in the whole slab cache (`cachep->lists.free_objects`) is greater than the limit stored in the `cachep->free_limit` field, then the function releases the slab's page frame(s) to the zoned page frame allocator:
      ```
      cachep->lists.free_objects -= cachep->num;
      slab_destroy(cachep, slabp);
      ```
      The value stored in the `cachep->free_limit` field is usually equal to `cachep->num + (1+N) × cachep->batchcount`, where N denotes the number of CPUs of the system.

   h. Otherwise, if `slab->inuse` is equal to zero but the number of free objects in the whole slab cache is less than `cachep->free_limit`, it inserts the slab descriptor in the `cachep->lists.slabs_free` list.

   i. Finally, if `slab->inuse` is greater than zero, the slab is partially filled, so the function inserts the slab descriptor in the `cachep->lists.slabs_partial` list.

4. Releases the `cachep->spinlock` spin lock.

5. Updates the `avail` field of the local cache descriptor by subtracting the number of objects moved to the shared local cache or released to the slab allocator.

6. Moves all valid pointers in the local cache at the beginning of the local cache's array. This step is necessary because the first object pointers have been removed from the local cache, thus the remaining ones must be moved up.

General Purpose Objects

As stated earlier in the section "The Buddy System Algorithm," infrequent requests for memory areas are handled through a group of general caches whose objects have geometrically distributed sizes ranging from a minimum of 32 to a maximum of 131,072 bytes.

Objects of this type are obtained by invoking the `kmalloc( )` function, which is essentially equivalent to the following code fragment:

```
void * kmalloc(size_t size, int flags)
{
    struct cache_sizes *csizep = malloc_sizes;
    kmem_cache_t * cachep;
    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        if (flags & __GFP_DMA)
            cachep = csizep->cs_dmacachep;
        else
            cachep = csizep->cs_cachep;
        return kmem_cache_alloc(cachep, flags);
    }
    return NULL;
}
```

The function uses the `malloc_sizes` table to locate the nearest power-of-2 size to the requested size. It then calls `kmem_cache_alloc( )` to allocate the object, passing to it either the cache descriptor for the page frames usable for ISA DMA or the cache descriptor for the "normal" page frames, depending on whether the caller specified the `__GFP_DMA` flag.

Objects obtained by invoking `kmalloc( )` can be released by calling `kfree( )`:

```
void kfree(const void *objp)
{
    kmem_cache_t * c;
    unsigned long flags;
    if (!objp)
        return;
    local_irq_save(flags);
    c = (kmem_cache_t *)(virt_to_page(objp)->lru.next);
    kmem_cache_free(c, (void *)objp);
    local_irq_restore(flags);
}
```

The proper cache descriptor is identified by reading the `lru.next` subfield of the descriptor of the first page frame containing the memory area. The memory area is released by invoking `kmem_cache_free( )`.

Memory Pools

Memory pools are a new feature of Linux 2.6. Basically, a memory pool allows a kernel component—such as the block device subsystem—to allocate some dynamic memory to be used only in low-on-memory emergencies.

Memory pools should not be confused with the reserved page frames described in the earlier section "The Pool of Reserved Page Frames." In fact, those page frames can be used only to satisfy atomic memory allocation requests issued by interrupt handlers or inside critical regions. Instead, a memory pool is a reserve of dynamic memory that can be used only by a specific kernel component, namely the "owner" of the pool. The owner does not normally use the reserve; however, if dynamic memory becomes so scarce that all usual memory allocation requests are doomed to fail, the kernel component can invoke, as a last resort, special memory pool functions that dip in the reserve and get the memory needed. Thus, creating a memory pool is similar to keeping a reserve of canned foods on hand and using a can opener only when no fresh food is available.

Often, a memory pool is stacked over the slab allocator—that is, it is used to keep a reserve of slab objects. Generally speaking, however, a memory pool can be used to allocate every kind of dynamic memory, from whole page frames to small memory areas allocated with `kmalloc()`. Therefore, we will generically refer to the memory units handled by a memory pool as "memory elements."

A memory pool is described by a `mempool_t` object, whose fields are shown in Table 8-12.

Table 8-12. The fields of the mempool_t object

Type	Name	Description
spinlock_t	lock	Spin lock protecting the object fields
int	min_nr	Maximum number of elements in the memory pool
int	curr_nr	Current number of elements in the memory pool
void **	elements	Pointer to an array of pointers to the reserved elements
void *	pool_data	Private data available to the pool's owner
mempool_alloc_t *	alloc	Method to allocate an element
mempool_free_t *	free	Method to free an element
wait_queue_head_t	wait	Wait queue used when the memory pool is empty

The `min_nr` field stores the initial number of elements in the memory pool. In other words, the value stored in this field represents the number of memory elements that

the owner of the memory pool is sure to obtain from the memory allocator. The `curr_nr` field, which is always lower than or equal to `min_nr`, stores the number of memory elements currently included in the memory pool. The memory elements themselves are referenced by an array of pointers, whose address is stored in the `elements` field.

The `alloc` and `free` methods interface with the underlying memory allocator to get and release a memory element, respectively. Both methods may be custom functions provided by the kernel component that owns the memory pool.

When the memory elements are slab objects, the `alloc` and `free` methods are commonly implemented by the `mempool_alloc_slab()` and `mempool_free_slab()` functions, which just invoke the `kmem_cache_alloc()` and `kmem_cache_free()` functions, respectively. In this case, the `pool_data` field of the `mempool_t` object stores the address of the slab cache descriptor.

The `mempool_create()` function creates a new memory pool; it receives the number of memory elements `min_nr`, the addresses of the functions that implement the `alloc` and `free` methods, and an optional value for the `pool_data` field. The function allocates memory for the `mempool_t` object and the array of pointers to the memory elements, then repeatedly invokes the `alloc` method to get the `min_nr` memory elements. Conversely, the `mempool_destroy()` function releases all memory elements in the pool, then releases the array of elements and the `mempool_t` object themselves.

To allocate an element from a memory pool, the kernel invokes the `mempool_alloc()` function, passing to it the address of the `mempool_t` object and the memory allocation flags (see Table 8-5 and Table 8-6 earlier in this chapter). Essentially, the function tries to allocate a memory element from the underlying memory allocator by invoking the `alloc` method, according to the memory allocation flags specified as parameters. If the allocation succeeds, the function returns the memory element obtained, without touching the memory pool. Otherwise, if the allocation fails, the memory element is taken from the memory pool. Of course, too many allocations in a low-on-memory condition can exhaust the memory pool: in this case, if the `__GFP_WAIT` flag is not set, `mempool_alloc()` blocks the current process until a memory element is released to the memory pool.

Conversely, to release an element to a memory pool, the kernel invokes the `mempool_free()` function. If the memory pool is not full (`curr_min` is smaller than `min_nr`), the function adds the element to the memory pool. Otherwise, `mempool_free()` invokes the `free` method to release the element to the underlying memory allocator.

Noncontiguous Memory Area Management

We already know that it is preferable to map memory areas into sets of contiguous page frames, thus making better use of the cache and achieving lower average memory access times. Nevertheless, if the requests for memory areas are infrequent, it

makes sense to consider an allocation scheme based on noncontiguous page frames accessed through contiguous linear addresses. The main advantage of this schema is to avoid external fragmentation, while the disadvantage is that it is necessary to fiddle with the kernel Page Tables. Clearly, the size of a noncontiguous memory area must be a multiple of 4,096. Linux uses noncontiguous memory areas in several ways—for instance, to allocate data structures for active swap areas (see the section "Activating and Deactivating a Swap Area" in Chapter 17), to allocate space for a module (see Appendix B), or to allocate buffers to some I/O drivers. Furthermore, noncontiguous memory areas provide yet another way to make use of high memory page frames (see the later section "Allocating a Noncontiguous Memory Area").

Linear Addresses of Noncontiguous Memory Areas

To find a free range of linear addresses, we can look in the area starting from PAGE_OFFSET (usually 0xc0000000, the beginning of the fourth gigabyte). Figure 8-7 shows how the fourth gigabyte linear addresses are used:

- The beginning of the area includes the linear addresses that map the first 896 MB of RAM (see the section "Process Page Tables" in Chapter 2); the linear address that corresponds to the end of the directly mapped physical memory is stored in the high_memory variable.

- The end of the area contains the fix-mapped linear addresses (see the section "Fix-Mapped Linear Addresses" in Chapter 2).

- Starting from PKMAP_BASE we find the linear addresses used for the persistent kernel mapping of high-memory page frames (see the section "Kernel Mappings of High-Memory Page Frames" earlier in this chapter).

- The remaining linear addresses can be used for noncontiguous memory areas. A safety interval of size 8 MB (macro VMALLOC_OFFSET) is inserted between the end of the physical memory mapping and the first memory area; its purpose is to "capture" out-of-bounds memory accesses. For the same reason, additional safety intervals of size 4 KB are inserted to separate noncontiguous memory areas.
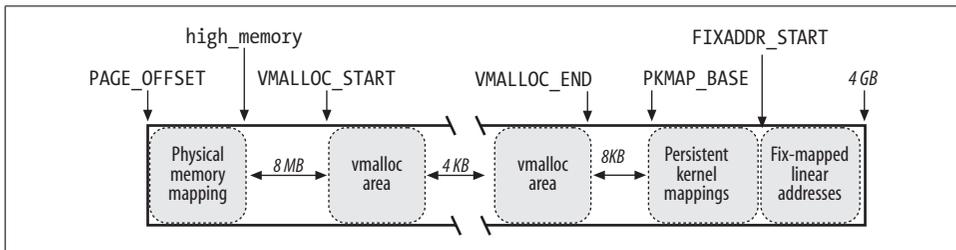
Figure 8-7. The linear address interval starting from PAGE_OFFSET

The `VMALLOC_START` macro defines the starting address of the linear space reserved for noncontiguous memory areas, while `VMALLOC_END` defines its ending address.

Descriptors of Noncontiguous Memory Areas

Each noncontiguous memory area is associated with a descriptor of type `vm_struct`, whose fields are listed in Table 8-13.

Table 8-13. The fields of the vm_struct descriptor

Type	Name	Description
void *	addr	Linear address of the first memory cell of the area
unsigned long	size	Size of the area plus 4,096 (inter-area safety interval)
unsigned long	flags	Type of memory mapped by the noncontiguous memory area
struct page **	pages	Pointer to array of nr_pages pointers to page descriptors
unsigned int	nr_pages	Number of pages filled by the area
unsigned long	phys_addr	Set to 0 unless the area has been created to map the I/O shared memory of a hardware device
struct vm_struct *	next	Pointer to next vm_struct structure

These descriptors are inserted in a simple list by means of the `next` field; the address of the first element of the list is stored in the `vmlist` variable. Accesses to this list are protected by means of the `vmlist_lock` read/write spin lock. The `flags` field identifies the type of memory mapped by the area: `VM_ALLOC` for pages obtained by means of `vmalloc()`, `VM_MAP` for already allocated pages mapped by means of `vmap()` (see the next section), and `VM_IOREMAP` for on-board memory of hardware devices mapped by means of `ioremap()` (see Chapter 13).

The `get_vm_area()` function looks for a free range of linear addresses between `VMALLOC_START` and `VMALLOC_END`. This function acts on two parameters: the size (`size`) in bytes of the memory region to be created, and a flag (`flag`) specifying the type of region (see above). The steps performed are the following:

1. Invokes `kmalloc()` to obtain a memory area for the new descriptor of type `vm_struct`.

2. Gets the `vmlist_lock` lock for writing and scans the list of descriptors of type `vm_struct` looking for a free range of linear addresses that includes at least `size+4096` addresses (4096 is the size of the safety interval between the memory areas).

3. If such an interval exists, the function initializes the fields of the descriptor, releases the `vmlist_lock` lock, and terminates by returning the initial address of the noncontiguous memory area.

4. Otherwise, `get_vm_area()` releases the descriptor obtained previously, releases the `vmlist_lock` lock, and returns `NULL`.

Allocating a Noncontiguous Memory Area

The vmalloc() function allocates a noncontiguous memory area to the kernel. The parameter size denotes the size of the requested area. If the function is able to satisfy the request, it then returns the initial linear address of the new area; otherwise, it returns a NULL pointer:

```
void * vmalloc(unsigned long size)
{
    struct vm_struct *area;
    struct page **pages;
    unsigned int array_size, i;
    size = (size + PAGE_SIZE - 1) & PAGE_MASK;
    area = get_vm_area(size, VM_ALLOC);
    if (!area)
        return NULL;
    area->nr_pages = size >> PAGE_SHIFT;
    array_size = (area->nr_pages * sizeof(struct page *));
    area->pages = pages = kmalloc(array_size, GFP_KERNEL);
    if (!area_pages) {
        remove_vm_area(area->addr);
        kfree(area);
        return NULL;
    }
    memset(area->pages, 0, array_size);
    for (i=0; i<area->nr_pages; i++) {
        area->pages[i] = alloc_page(GFP_KERNEL|__GFP_HIGHMEM);
        if (!area->pages[i]) {
            area->nr_pages = i;
fail:       vfree(area->addr);
            return NULL;
        }
    }
    if (map_vm_area(area, __pgprot(0x63), &pages))
        goto fail;
    return area->addr;
}
```

The function starts by rounding up the value of the size parameter to a multiple of 4,096 (the page frame size). Then vmalloc() invokes get_vm_area(), which creates a new descriptor and returns the linear addresses assigned to the memory area. The flags field of the descriptor is initialized with the VM_ALLOC flag, which means that the noncontiguous page frames will be mapped into a linear address range by means of the vmalloc() function. Then the vmalloc() function invokes kmalloc() to request a group of contiguous page frames large enough to contain an array of page descriptor pointers. The memset() function is invoked to set all these pointers to NULL. Next the alloc_page() function is called repeatedly, once for each of the nr_pages of the region, to allocate a page frame and store the address of the corresponding page descriptor in the area->pages array. Observe that using the area->pages array is necessary because the page frames could belong to the ZONE_HIGHMEM memory zone, thus right now they are not necessarily mapped to a linear address.

Now comes the tricky part. Up to this point, a fresh interval of contiguous linear addresses has been obtained and a group of noncontiguous page frames has been allocated to map these linear addresses. The last crucial step consists of fiddling with the page table entries used by the kernel to indicate that each page frame allocated to the noncontiguous memory area is now associated with a linear address included in the interval of contiguous linear addresses yielded by vmalloc(). This is what map_vm_area() does.

The map_vm_area() function uses three parameters:

area

> The pointer to the vm_struct descriptor of the area.

prot

> The protection bits of the allocated page frames. It is always set to 0x63, which corresponds to Present, Accessed, Read/Write, and Dirty.

pages

> The address of a variable pointing to an array of pointers to page descriptors (thus, struct page *** is used as the data type!).

The function starts by assigning the linear addresses of the start and end of the area to the address and end local variables, respectively:

```
address = area->addr;
end = address + (area->size - PAGE_SIZE);
```

Remember that area->size stores the actual size of the area plus the 4 KB inter-area safety interval. The function then uses the pgd_offset_k macro to derive the entry in the master kernel Page Global Directory related to the initial linear address of the area; it then acquires the kernel Page Table spin lock:

```
pgd = pgd_offset_k(address);
spin_lock(&init_mm.page_table_lock);
```

The function then executes the following cycle:

```
int ret = 0;
for (i = pgd_index(address); i < pgd_index(end-1); i++) {
    pud_t *pud = pud_alloc(&init_mm, pgd, address);
    ret = -ENOMEM;
    if (!pud)
        break;
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next < address || next > end)
        next = end;
    if (map_area_pud(pud, address, next, prot, pages))
        break;
    address = next;
    pgd++;
    ret = 0;
}
spin_unlock(&init_mm.page_table_lock);
```

```
        flush_cache_vmap((unsigned long)area->addr, end);
        return ret;
```

In each cycle, it first invokes pud_alloc( ) to create a Page Upper Directory for the new area and writes its physical address in the right entry of the kernel Page Global Directory. It then calls map_area_pud( ) to allocate all the page tables associated with the new Page Upper Directory. It adds the size of the range of linear addresses spanned by a single Page Upper Directory—the constant 2^{30} if PAE is enabled, 2^{22} otherwise—to the current value of address, and it increases the pointer pgd to the Page Global Directory.

The cycle is repeated until all Page Table entries referring to the noncontiguous memory area are set up.

The map_area_pud( ) function executes a similar cycle for all the page tables that a Page Upper Directory points to:

```
do {
    pmd_t * pmd = pmd_alloc(&init_mm, pud, address);
    if (!pmd)
        return -ENOMEM;
    if (map_area_pmd(pmd, address, end-address, prot, pages))
        return -ENOMEM;
    address = (address + PUD_SIZE) & PUD_MASK;
    pud++;
} while (address < end);
```

The map_area_pmd( ) function executes a similar cycle for all the Page Tables that a Page Middle Directory points to:

```
do {
    pte_t * pte = pte_alloc_kernel(&init_mm, pmd, address);
    if (!pte)
        return -ENOMEM;
    if (map_area_pte(pte, address, end-address, prot, pages))
        return -ENOMEM;
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
} while (address < end);
```

The pte_alloc_kernel( ) function (see the section "Page Table Handling" in Chapter 2) allocates a new Page Table and updates the corresponding entry in the Page Middle Directory. Next, map_area_pte( ) allocates all the page frames corresponding to the entries in the Page Table. The value of address is increased by 2^{22} — the size of the linear address interval spanned by a single Page Table—and the cycle is repeated.

The main cycle of map_area_pte( ) is:

```
do {
    struct page * page = **pages;
    set_pte(pte, mk_pte(page, prot));
    address += PAGE_SIZE;
```

```
    pte++;
    (*pages)++;
} while (address < end);
```

The page descriptor address page of the page frame to be mapped is read from the array's entry pointed to by the variable at address pages. The physical address of the new page frame is written into the Page Table by the set_pte and mk_pte macros. The cycle is repeated after adding the constant 4,096 (the length of a page frame) to address.

Notice that the Page Tables of the current process are not touched by map_vm_area(). Therefore, when a process in Kernel Mode accesses the noncontiguous memory area, a Page Fault occurs, because the entries in the process's Page Tables corresponding to the area are null. However, the Page Fault handler checks the faulty linear address against the master kernel Page Tables (which are init_mm.pgd Page Global Directory and its child page tables; see the section "Kernel Page Tables" in Chapter 2). Once the handler discovers that a master kernel Page Table includes a non-null entry for the address, it copies its value into the corresponding process's Page Table entry and resumes normal execution of the process. This mechanism is described in the section "Page Fault Exception Handler" in Chapter 9.

Beside the vmalloc() function, a noncontiguous memory area can be allocated by the vmalloc_32() function, which is very similar to vmalloc() but only allocates page frames from the ZONE_NORMAL and ZONE_DMA memory zones.

Linux 2.6 also features a vmap() function, which maps page frames already allocated in a noncontiguous memory area: essentially, this function receives as its parameter an array of pointers to page descriptors, invokes get_vm_area() to get a new vm_struct descriptor, and then invokes map_vm_area() to map the page frames. The function is thus similar to vmalloc(), but it does not allocate page frames.

Releasing a Noncontiguous Memory Area

The vfree() function releases noncontiguous memory areas created by vmalloc() or vmalloc_32(), while the vunmap() function releases memory areas created by vmap(). Both functions have one parameter—the address of the initial linear address of the area to be released; they both rely on the __vunmap() function to do the real work.

The __vunmap() function receives two parameters: the address addr of the initial linear address of the area to be released, and the flag deallocate_pages, which is set if the page frames mapped in the area should be released to the zoned page frame allocator (vfree()'s invocation), and cleared otherwise (vunmap()'s invocation). The function performs the following operations:

1. Invokes the remove_vm_area() function to get the address area of the vm_struct descriptor and to clear the kernel's page table entries corresponding to the linear address in the noncontiguous memory area.

---

2. If the deallocate_pages flag is set, it scans the area->pages array of pointers to the page descriptor; for each element of the array, invokes the __free_page() function to release the page frame to the zoned page frame allocator. Moreover, executes kfree(area->pages) to release the array itself.

3. Invokes kfree(area) to release the vm_struct descriptor.

The remove_vm_area( ) function performs the following cycle:

```
write_lock(&vmlist_lock);
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        unmap_vm_area(tmp);
        *p = tmp->next;
        break;
    }
}
write_unlock(&vmlist_lock);
return tmp;
```

The area itself is released by invoking unmap_vm_area( ). This function acts on a single parameter, namely a pointer area to the vm_struct descriptor of the area. It executes the following cycle to reverse the actions performed by map_vm_area( ):

```
address = area->addr;
end = address + area->size;
pgd = pgd_offset_k(address);
for (i = pgd_index(address); i <= pgd_index(end-1); i++) {
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next <= address || next > end)
        next = end;
    unmap_area_pud(pgd, address, next - address);
    address = next;
    pgd++;
}
```

In turn, unmap_area_pud( ) reverses the actions of map_area_pud( ) in the cycle:

```
do {
    unmap_area_pmd(pud, address, end-address);
    address = (address + PUD_SIZE) & PUD_MASK;
    pud++;
} while (address && (address < end));
```

The unmap_area_pmd( ) function reverses the actions of map_area_pmd( ) in the cycle:

```
do {
    unmap_area_pte(pmd, address, end-address);
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
} while (address < end);
```

Finally, unmap_area_pte( ) reverses the actions of map_area_pte( ) in the cycle:

```
do {
    pte_t page = ptep_get_and_clear(pte);
```

```
        address += PAGE_SIZE;
        pte++;
        if (!pte_none(page) && !pte_present(page))
            printk("Whee... Swapped out page in kernel page table\n");
    } while (address < end);
```

In every iteration of the cycle, the page table entry pointed to by `pte` is set to 0 by the `ptep_get_and_clear` macro.

As for `vmalloc()`, the kernel modifies the entries of the master kernel Page Global Directory and its child page tables (see the section "Kernel Page Tables" in Chapter 2), but it leaves unchanged the entries of the process page tables mapping the fourth gigabyte. This is fine because the kernel never reclaims Page Upper Directories, Page Middle Directories, and Page Tables rooted at the master kernel Page Global Directory.

For instance, suppose that a process in Kernel Mode accessed a noncontiguous memory area that later got released. The process's Page Global Directory entries are equal to the corresponding entries of the master kernel Page Global Directory, thanks to the mechanism explained in the section "Page Fault Exception Handler" in Chapter 9; they point to the same Page Upper Directories, Page Middle Directories, and Page Tables. The `unmap_area_pte( )` function clears only the entries of the page tables (without reclaiming the page tables themselves). Further accesses of the process to the released noncontiguous memory area will trigger Page Faults because of the null page table entries. However, the handler will consider such accesses a bug, because the master kernel page tables do not include valid entries.

Process Address Space

As seen in the previous chapter, a kernel function gets dynamic memory in a fairly straightforward manner by invoking one of a variety of functions: `__get_free_pages()` or `alloc_pages()` to get pages from the zoned page frame allocator, `kmem_cache_alloc()` or `kmalloc()` to use the slab allocator for specialized or general-purpose objects, and `vmalloc()` or `vmalloc_32()` to get a noncontiguous memory area. If the request can be satisfied, each of these functions returns a page descriptor address or a linear address identifying the beginning of the allocated dynamic memory area.

These simple approaches work for two reasons:

- The kernel is the highest-priority component of the operating system. If a kernel function makes a request for dynamic memory, it must have a valid reason to issue that request, and there is no point in trying to defer it.

- The kernel trusts itself. All kernel functions are assumed to be error-free, so the kernel does not need to insert any protection against programming errors.

When allocating memory to User Mode processes, the situation is entirely different:

- Process requests for dynamic memory are considered non-urgent. When a process's executable file is loaded, for instance, it is unlikely that the process will address all the pages of code in the near future. Similarly, when a process invokes `malloc()` to get additional dynamic memory, it doesn't mean the process will soon access all the additional memory obtained. Thus, as a general rule, the kernel tries to defer allocating dynamic memory to User Mode processes.

- Because user programs cannot be trusted, the kernel must be prepared to catch all addressing errors caused by processes in User Mode.

As this chapter describes, the kernel succeeds in deferring the allocation of dynamic memory to processes by using a new kind of resource. When a User Mode process asks for dynamic memory, it doesn't get additional page frames; instead, it gets the right to use a new range of linear addresses, which become part of its address space. This interval is called a "memory region."

In the next section, we discuss how the process views dynamic memory. We then describe the basic components of the process address space in the section "Memory Regions." Next, we examine in detail the role played by the Page Fault exception handler in deferring the allocation of page frames to processes and illustrate how the kernel creates and deletes whole process address spaces. Last, we discuss the APIs and system calls related to address space management.

The Process's Address Space

The *address space* of a process consists of all linear addresses that the process is allowed to use. Each process sees a different set of linear addresses; the address used by one process bears no relation to the address used by another. As we will see later, the kernel may dynamically modify a process address space by adding or removing intervals of linear addresses.

The kernel represents intervals of linear addresses by means of resources called *memory regions*, which are characterized by an initial linear address, a length, and some access rights. For reasons of efficiency, both the initial address and the length of a memory region must be multiples of 4,096, so that the data identified by each memory region completely fills up the page frames allocated to it. Following are some typical situations in which a process gets new memory regions:

- When the user types a command at the console, the shell process creates a new process to execute the command. As a result, a fresh address space, and thus a set of memory regions, is assigned to the new process (see the section "Creating and Deleting a Process Address Space" later in this chapter; also, see Chapter 20).

- A running process may decide to load an entirely different program. In this case, the process ID remains unchanged, but the memory regions used before loading the program are released and a new set of memory regions is assigned to the process (see the section "The exec Functions" in Chapter 20).

- A running process may perform a "memory mapping" on a file (or on a portion of it). In such cases, the kernel assigns a new memory region to the process to map the file (see the section "Memory Mapping" in Chapter 16).

- A process may keep adding data on its User Mode stack until all addresses in the memory region that map the stack have been used. In this case, the kernel may decide to expand the size of that memory region (see the section "Page Fault Exception Handler" later in this chapter).

- A process may create an IPC-shared memory region to share data with other cooperating processes. In this case, the kernel assigns a new memory region to the process to implement this construct (see the section "IPC Shared Memory" in Chapter 19).

- A process may expand its dynamic area (the heap) through a function such as `malloc( )`. As a result, the kernel may decide to expand the size of the memory

region assigned to the heap (see the section "Managing the Heap" later in this chapter).

Table 9-1 illustrates some of the system calls related to the previously mentioned tasks. `brk( )` is discussed at the end of this chapter, while the remaining system calls are described in other chapters.

Table 9-1. System calls related to memory region creation and deletion

System call	Description
brk( )	Changes the heap size of the process
execve( )	Loads a new executable file, thus changing the process address space
_exit( )	Terminates the current process and destroys its address space
fork( )	Creates a new process, and thus a new address space
mmap( ), mmap2( )	Creates a memory mapping for a file, thus enlarging the process address space
mremap( )	Expands or shrinks a memory region
remap_file_pages( )	Creates a non-linear mapping for a file (see Chapter 16)
munmap( )	Destroys a memory mapping for a file, thus contracting the process address space
shmat( )	Attaches a shared memory region
shmdt( )	Detaches a shared memory region

As we'll see in the later section "Page Fault Exception Handler," it is essential for the kernel to identify the memory regions currently owned by a process (the address space of a process), because that allows the Page Fault exception handler to efficiently distinguish between two types of invalid linear addresses that cause it to be invoked:

- Those caused by programming errors.
- Those caused by a missing page; even though the linear address belongs to the process's address space, the page frame corresponding to that address has yet to be allocated.

The latter addresses are not invalid from the process's point of view; the induced Page Faults are exploited by the kernel to implement demand paging: the kernel provides the missing page frame and lets the process continue.

The Memory Descriptor

All information related to the process address space is included in an object called the *memory descriptor* of type `mm_struct`. This object is referenced by the `mm` field of the process descriptor. The fields of a memory descriptor are listed in Table 9-2.

Table 9-2. The fields of the memory descriptor

Type	Field	Description
struct vm_area_struct *	mmap	Pointer to the head of the list of memory region objects
struct rb_root	mm_rb	Pointer to the root of the red-black tree of memory region objects
struct vm_area_struct *	mmap_cache	Pointer to the last referenced memory region object
unsigned long (*)()	get_unmapped_area	Method that searches an available linear address interval in the process address space
void (*)()	unmap_area	Method invoked when releasing a linear address interval
unsigned long	mmap_base	Identifies the linear address of the first allocated anonymous memory region or file memory mapping (see the section "Program Segments and Process Memory Regions" in Chapter 20)
unsigned long	free_area_cache	Address from which the kernel will look for a free interval of linear addresses in the process address space
pgd_t *	pgd	Pointer to the Page Global Directory
atomic_t	mm_users	Secondary usage counter
atomic_t	mm_count	Main usage counter
int	map_count	Number of memory regions
struct rw_semaphore	mmap_sem	Memory regions' read/write semaphore
spinlock_t	page_table_lock	Memory regions' and Page Tables' spin lock
struct list_head	mmlist	Pointers to adjacent elements in the list of memory descriptors
unsigned long	start_code	Initial address of executable code
unsigned long	end_code	Final address of executable code
unsigned long	start_data	Initial address of initialized data
unsigned long	end_data	Final address of initialized data
unsigned long	start_brk	Initial address of the heap
unsigned long	brk	Current final address of the heap
unsigned long	start_stack	Initial address of User Mode stack
unsigned long	arg_start	Initial address of command-line arguments
unsigned long	arg_end	Final address of command-line arguments
unsigned long	env_start	Initial address of environment variables
unsigned long	env_end	Final address of environment variables
unsigned long	rss	Number of page frames allocated to the process
unsigned long	anon_rss	Number of page frames assigned to anonymous memory mappings

Table 9-2. The fields of the memory descriptor (continued)

Type	Field	Description
unsigned long	total_vm	Size of the process address space (number of pages)
unsigned long	locked_vm	Number of "locked" pages that cannot be swapped out (see Chapter 17)
unsigned long	shared_vm	Number of pages in shared file memory mappings
unsigned long	exec_vm	Number of pages in executable memory mappings
unsigned long	stack_vm	Number of pages in the User Mode stack
unsigned long	reserved_vm	Number of pages in reserved or special memory regions
unsigned long	def_flags	Default access flags of the memory regions
unsigned long	nr_ptes	Number of Page Tables of this process
unsigned long []	saved_auxv	Used when starting the execution of an ELF program (see Chapter 20)
unsigned int	dumpable	Flag that specifies whether the process can produce a core dump of the memory
cpumask_t	cpu_vm_mask	Bit mask for lazy TLB switches (see Chapter 2)
mm_context_t	context	Pointer to table for architecture-specific information (e.g., LDT's address in 80×86 platforms)
unsigned long	swap_token_time	When this process will become eligible for having the swap token (see the section "The Swap Token" in Chapter 17)
char	recent_pagein	Flag set if a major Page Fault has recently occurred
int	core_waiters	Number of lightweight processes that are dumping the contents of the process address space to a core file (see the section "Deleting a Process Address Space" later in this chapter)
struct completion *	core_startup_done	Pointer to a completion used when creating a core file (see the section "Completions" in Chapter 5)
struct completion	core_done	Completion used when creating a core file
rwlock_t	ioctx_list_lock	Lock used to protect the list of asynchronous I/O contexts (see Chapter 16)
struct kioctx *	ioctx_list	List of asynchronous I/O contexts (see Chapter 16)
struct kioctx	default_kioctx	Default asynchronous I/O context (see Chapter 16)
unsigned long	hiwater_rss	Maximum number of page frames ever owned by the process
unsigned long	hiwater_vm	Maximum number of pages ever included in the memory regions of the process

All memory descriptors are stored in a doubly linked list. Each descriptor stores the address of the adjacent list items in the mmlist field. The first element of the list is the mmlist field of init_mm, the memory descriptor used by process 0 in the initialization

phase. The list is protected against concurrent accesses in multiprocessor systems by the `mmlist_lock` spin lock.

The `mm_users` field stores the number of lightweight processes that share the `mm_struct` data structure (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3). The `mm_count` field is the main usage counter of the memory descriptor; all "users" in `mm_users` count as one unit in `mm_count`. Every time the `mm_count` field is decreased, the kernel checks whether it becomes zero; if so, the memory descriptor is deallocated because it is no longer in use.

We'll try to explain the difference between the use of `mm_users` and `mm_count` with an example. Consider a memory descriptor shared by two lightweight processes. Normally, its `mm_users` field stores the value 2, while its `mm_count` field stores the value 1 (both owner processes count as one).

If the memory descriptor is temporarily lent to a kernel thread (see the next section), the kernel increases the `mm_count` field. In this way, even if both lightweight processes die and the `mm_users` field becomes zero, the memory descriptor is not released until the kernel thread finishes using it because the `mm_count` field remains greater than zero.

If the kernel wants to be sure that the memory descriptor is not released in the middle of a lengthy operation, it might increase the `mm_users` field instead of `mm_count` (this is what the `try_to_unuse( )` function does; see the section "Activating and Deactivating a Swap Area" in Chapter 17). The final result is the same because the increment of `mm_users` ensures that `mm_count` does not become zero even if all lightweight processes that own the memory descriptor die.

The `mm_alloc( )` function is invoked to get a new memory descriptor. Because these descriptors are stored in a slab allocator cache, `mm_alloc( )` calls `kmem_cache_alloc( )`, initializes the new memory descriptor, and sets the `mm_count` and `mm_users` field to 1.

Conversely, the `mmput( )` function decreases the `mm_users` field of a memory descriptor. If that field becomes 0, the function releases the Local Descriptor Table, the memory region descriptors (see later in this chapter), and the Page Tables referenced by the memory descriptor, and then invokes `mmdrop( )`. The latter function decreases `mm_count` and, if it becomes zero, releases the `mm_struct` data structure.

The `mmap`, `mm_rb`, `mmlist`, and `mmap_cache` fields are discussed in the next section.

Memory Descriptor of Kernel Threads

Kernel threads run only in Kernel Mode, so they never access linear addresses below `TASK_SIZE` (same as `PAGE_OFFSET`, usually `0xc0000000`). Contrary to regular processes, kernel threads do not use memory regions, therefore most of the fields of a memory descriptor are meaningless for them.

Because the Page Table entries that refer to the linear address above `TASK_SIZE` should always be identical, it does not really matter what set of Page Tables a kernel thread uses. To avoid useless TLB and cache flushes, a kernel thread uses the set of Page Tables of the last previously running regular process. To that end, two kinds of memory descriptor pointers are included in every process descriptor: `mm` and `active_mm`.

The `mm` field in the process descriptor points to the memory descriptor owned by the process, while the `active_mm` field points to the memory descriptor used by the process when it is in execution. For regular processes, the two fields store the same pointer. Kernel threads, however, do not own any memory descriptor, thus their `mm` field is always `NULL`. When a kernel thread is selected for execution, its `active_mm` field is initialized to the value of the `active_mm` of the previously running process (see the section "The schedule( ) Function" in Chapter 7).

There is, however, a small complication. Whenever a process in Kernel Mode modifies a Page Table entry for a "high" linear address (above `TASK_SIZE`), it should also update the corresponding entry in the sets of Page Tables of all processes in the system. In fact, once set by a process in Kernel Mode, the mapping should be effective for all other processes in Kernel Mode as well. Touching the sets of Page Tables of all processes is a costly operation; therefore, Linux adopts a deferred approach.

We already mentioned this deferred approach in the section "Noncontiguous Memory Area Management" in Chapter 8: every time a high linear address has to be remapped (typically by `vmalloc( )` or `vfree( )`), the kernel updates a canonical set of Page Tables rooted at the `swapper_pg_dir` master kernel Page Global Directory (see the section "Kernel Page Tables" in Chapter 2). This Page Global Directory is pointed to by the `pgd` field of a *master memory descriptor*, which is stored in the `init_mm` variable.[*]

Later, in the section "Handling Noncontiguous Memory Area Accesses," we'll describe how the Page Fault handler takes care of spreading the information stored in the canonical Page Tables when effectively needed.

Memory Regions

Linux implements a memory region by means of an object of type `vm_area_struct`; its fields are shown in Table 9-3.[†]

---

[*] We mentioned in the section "Kernel Threads" in Chapter 3 that the *swapper* process uses `init_mm` during the initialization phase. However, swapper never uses this memory descriptor once the initialization phase completes.

[†] We omitted describing a few additional fields used in NUMA systems.

Table 9-3. The fields of the memory region object

Type	Field	Description
struct mm_struct *	vm_mm	Pointer to the memory descriptor that owns the region.
unsigned long	vm_start	First linear address inside the region.
unsigned long	vm_end	First linear address after the region.
struct vm_area_struct *	vm_next	Next region in the process list.
pgprot_t	vm_page_prot	Access permissions for the page frames of the region.
unsigned long	vm_flags	Flags of the region.
struct rb_node	vm_rb	Data for the red-black tree (see later in this chapter).
union	shared	Links to the data structures used for reverse mapping (see the section "Reverse Mapping for Mapped Pages" in Chapter 17).
struct list_head	anon_vma_node	Pointers for the list of anonymous memory regions (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
struct anon_vma *	anon_vma	Pointer to the anon_vma data structure (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
struct vm_operations_struct*	vm_ops	Pointer to the methods of the memory region.
unsigned long	vm_pgoff	Offset in mapped file (see Chapter 16). For anonymous pages, it is either zero or equal to vm_start/PAGE_SIZE (see Chapter 17).
struct file *	vm_file	Pointer to the file object of the mapped file, if any.
void *	vm_private_data	Pointer to private data of the memory region.
unsigned long	vm_truncate_count	Used when releasing a linear address interval in a non-linear file memory mapping.

Each memory region descriptor identifies a linear address interval. The vm_start field contains the first linear address of the interval, while the vm_end field contains the first linear address outside of the interval; vm_end–vm_start thus denotes the length of the memory region. The vm_mm field points to the mm_struct memory descriptor of the process that owns the region. We will describe the other fields of vm_area_struct as they come up.

Memory regions owned by a process never overlap, and the kernel tries to merge regions when a new one is allocated right next to an existing one. Two adjacent regions can be merged if their access rights match.

As shown in Figure 9-1, when a new range of linear addresses is added to the process address space, the kernel checks whether an already existing memory region can be enlarged (case *a*). If not, a new memory region is created (case *b*). Similarly, if a range of linear addresses is removed from the process address space, the kernel

resizes the affected memory regions (case *c*). In some cases, the resizing forces a memory region to split into two smaller ones (case *d*).*
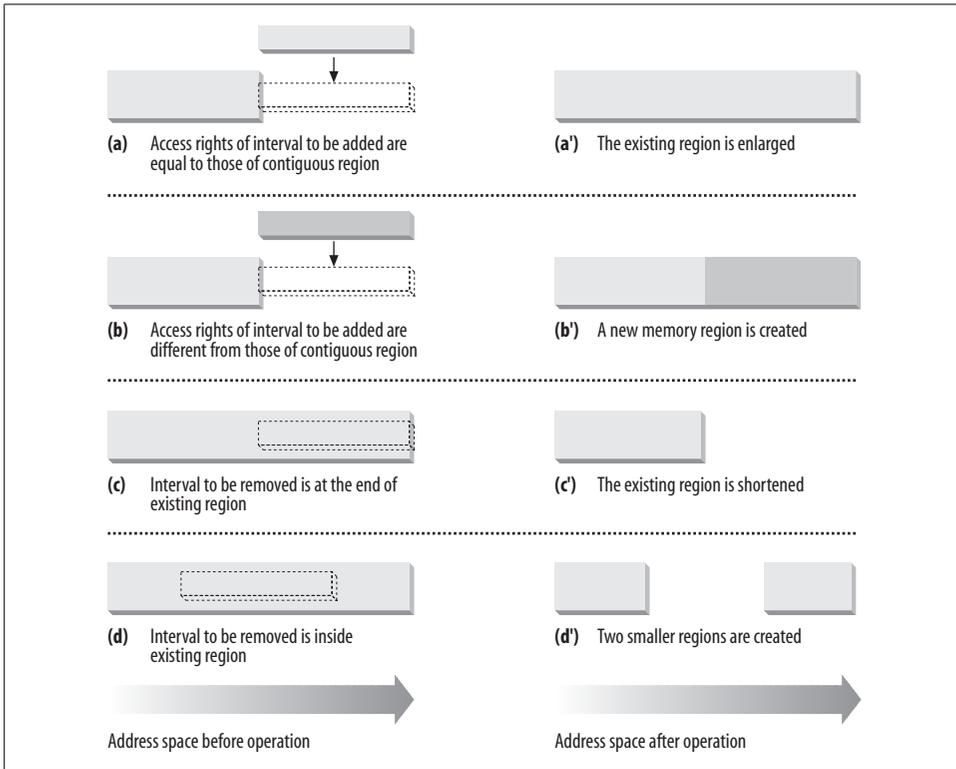
Figure 9-1. Adding or removing a linear address interval

The vm_ops field points to a vm_operations_struct data structure, which stores the methods of the memory region. Only four methods—illustrated in Table 9-4—are applicable to UMA systems.

Table 9-4. The methods to act on a memory region

Method	Description
open	Invoked when the memory region is added to the set of regions owned by a process.
close	Invoked when the memory region is removed from the set of regions owned by a process.
nopage	Invoked by the Page Fault exception handler when a process tries to access a page not present in RAM whose linear address belongs to the memory region (see the later section "Page Fault Exception Handler").

---

* Removing a linear address interval may theoretically fail because no free memory is available for a new memory descriptor.

Table 9-4. The methods to act on a memory region (continued)

Method	Description
populate	Invoked to set the page table entries corresponding to the linear addresses of the memory region (prefaulting). Mainly used for non-linear file memory mappings.

Memory Region Data Structures

All the regions owned by a process are linked in a simple list. Regions appear in the list in ascending order by memory address; however, successive regions can be separated by an area of unused memory addresses. The vm_next field of each vm_area_ struct element points to the next element in the list. The kernel finds the memory regions through the mmap field of the process memory descriptor, which points to the first memory region descriptor in the list.

The map_count field of the memory descriptor contains the number of regions owned by the process. By default, a process may own up to 65,536 different memory regions; however, the system administrator may change this limit by writing in the */proc/sys/vm/max_map_count* file.

Figure 9-2 illustrates the relationships among the address space of a process, its memory descriptor, and the list of memory regions.
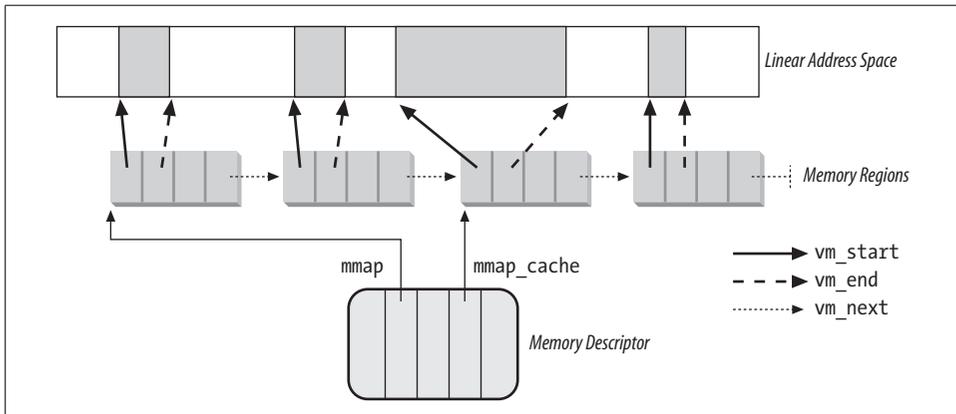
Figure 9-2. Descriptors related to the address space of a process

A frequent operation performed by the kernel is to search the memory region that includes a specific linear address. Because the list is sorted, the search can terminate as soon as a memory region that ends after the specific linear address is found.

However, using the list is convenient only if the process has very few memory regions—let's say less than a few tens of them. Searching, inserting elements, and deleting elements in the list involve a number of operations whose times are linearly proportional to the list length.

Although most Linux processes use very few memory regions, there are some large applications, such as object-oriented databases or specialized debuggers for the usage of malloc(), that have many hundreds or even thousands of regions. In such cases, the memory region list management becomes very inefficient, hence the performance of the memory-related system calls degrades to an intolerable point.

Therefore, Linux 2.6 stores memory descriptors in data structures called *red-black trees*. In a red-black tree, each element (or *node*) usually has two children: a *left child* and a *right child*. The elements in the tree are sorted. For each node N, all elements of the subtree rooted at the left child of N precede N, while, conversely, all elements of the subtree rooted at the right child of N follow N (see Figure 9-3(*a*); the key of the node is written inside the node itself. Moreover, a red-black tree must satisfy four additional rules:

1. Every node must be either red or black.
2. The root of the tree must be black.
3. The children of a red node must be black.
4. Every path from a node to a descendant leaf must contain the same number of black nodes. When counting the number of black nodes, null pointers are counted as black nodes.
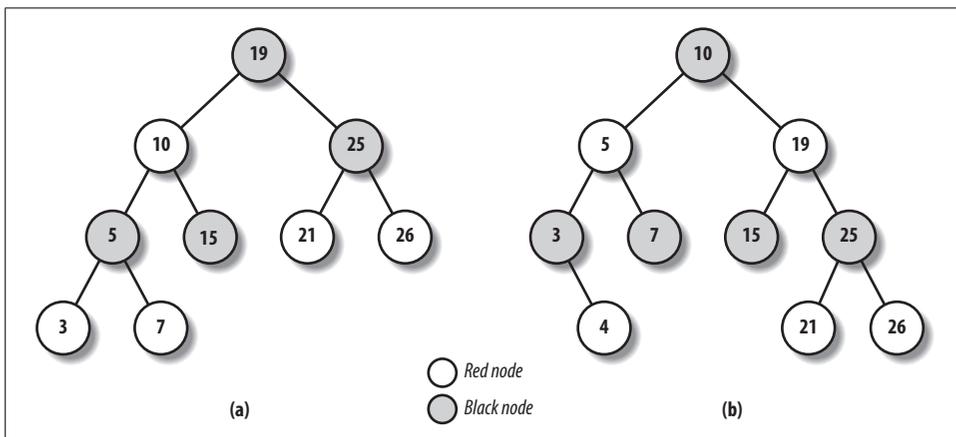
Figure 9-3. Example of red-black trees

These four rules ensure that every red-black tree with *n* internal nodes has a height of at most $2 \times \log(n + 1)$.

Searching an element in a red-black tree is thus very efficient, because it requires operations whose execution time is linearly proportional to the logarithm of the tree size. In other words, doubling the number of memory regions adds just one more iteration to the operation.

Inserting and deleting an element in a red-black tree is also efficient, because the algorithm can quickly traverse the tree to locate the position at which the element will be inserted or from which it will be removed. Each new node must be inserted as a leaf and colored red. If the operation breaks the rules, a few nodes of the tree must be moved or recolored.

For instance, suppose that an element having the value 4 must be inserted in the red-black tree shown in Figure 9-3(a). Its proper position is the right child of the node that has key 3, but once it is inserted, the red node that has the value 3 has a red child, thus breaking rule 3. To satisfy the rule, the color of nodes that have the values 3, 4, and 7 is changed. This operation, however, breaks rule 4, thus the algorithm performs a "rotation" on the subtree rooted at the node that has the key 19, producing the new red-black tree shown in Figure 9-3(b). This looks complicated, but inserting or deleting an element in a red-black tree requires a small number of operations—a number linearly proportional to the logarithm of the tree size.

Therefore, to store the memory regions of a process, Linux uses both a linked list and a red-black tree. Both data structures contain pointers to the same memory region descriptors. When inserting or removing a memory region descriptor, the kernel searches the previous and next elements through the red-black tree and uses them to quickly update the list without scanning it.

The head of the linked list is referenced by the `mmap` field of the memory descriptor. Each memory region object stores the pointer to the next element of the list in the `vm_next` field. The head of the red-black tree is referenced by the `mm_rb` field of the memory descriptor. Each memory region object stores the color of the node, as well as the pointers to the parent, the left child, and the right child, in the `vm_rb` field of type `rb_node`.

In general, the red-black tree is used to locate a region including a specific address, while the linked list is mostly useful when scanning the whole set of regions.

Memory Region Access Rights

Before moving on, we should clarify the relation between a page and a memory region. As mentioned in Chapter 2, we use the term "page" to refer both to a set of linear addresses and to the data contained in this group of addresses. In particular, we denote the linear address interval ranging between 0 and 4,095 as page 0, the linear address interval ranging between 4,096 and 8,191 as page 1, and so forth. Each memory region therefore consists of a set of pages that have consecutive page numbers.

We have already discussed two kinds of flags associated with a page:

- A few flags such as `Read/Write`, `Present`, or `User/Supervisor` stored in each Page Table entry (see the section "Regular Paging" in Chapter 2).
- A set of flags stored in the `flags` field of each `page` descriptor (see the section "Page Frame Management" in Chapter 8).

The first kind of flag is used by the 80×86 hardware to check whether the requested kind of addressing can be performed; the second kind is used by Linux for many different purposes (see Table 8-2).

We now introduce a third kind of flag: those associated with the pages of a memory region. They are stored in the vm_flags field of the vm_area_struct descriptor (see Table 9-5). Some flags offer the kernel information about all the pages of the memory region, such as what they contain and what rights the process has to access each page. Other flags describe the region itself, such as how it can grow.

Table 9-5. The memory region flags

Flag name	Description
VM_READ	Pages can be read
VM_WRITE	Pages can be written
VM_EXEC	Pages can be executed
VM_SHARED	Pages can be shared by several processes
VM_MAYREAD	VM_READ flag may be set
VM_MAYWRITE	VM_WRITE flag may be set
VM_MAYEXEC	VM_EXEC flag may be set
VM_MAYSHARE	VM_SHARE flag may be set
VM_GROWSDOWN	The region can expand toward lower addresses
VM_GROWSUP	The region can expand toward higher addresses
VM_SHM	The region is used for IPC's shared memory
VM_DENYWRITE	The region maps a file that cannot be opened for writing
VM_EXECUTABLE	The region maps an executable file
VM_LOCKED	Pages in the region are locked and cannot be swapped out
VM_IO	The region maps the I/O address space of a device
VM_SEQ_READ	The application accesses the pages sequentially
VM_RAND_READ	The application accesses the pages in a truly random order
VM_DONTCOPY	Do not copy the region when forking a new process
VM_DONTEXPAND	Forbid region expansion through mremap( ) system call
VM_RESERVED	The region is special (for instance, it maps the I/O address space of a device), so its pages must not be swapped out
VM_ACCOUNT	Check whether there is enough free memory for the mapping when creating an IPC shared memory region (see Chapter 19)
VM_HUGETLB	The pages in the region are handled through the extended paging mechanism (see the section "Extended Paging" in Chapter 2)
VM_NONLINEAR	The region implements a non-linear file mapping

Page access rights included in a memory region descriptor may be combined arbitrarily. It is possible, for instance, to allow the pages of a region to be read but not

executed. To implement this protection scheme efficiently, the Read, Write, and Execute access rights associated with the pages of a memory region must be duplicated in all the corresponding Page Table entries, so that checks can be directly performed by the Paging Unit circuitry. In other words, the page access rights dictate what kinds of access should generate a Page Fault exception. As we'll see shortly, the job of figuring out what caused the Page Fault is delegated by Linux to the Page Fault handler, which implements several page-handling strategies.

The initial values of the Page Table flags (which must be the same for all pages in the memory region, as we have seen) are stored in the `vm_page_prot` field of the `vm_area_struct` descriptor. When adding a page, the kernel sets the flags in the corresponding Page Table entry according to the value of the `vm_page_prot` field.

However, translating the memory region's access rights into the page protection bits is not straightforward for the following reasons:

- In some cases, a page access should generate a Page Fault exception even when its access type is granted by the page access rights specified in the `vm_flags` field of the corresponding memory region. For instance, as we'll see in the section "Copy On Write" later in this chapter, the kernel may wish to store two identical, writable private pages (whose `VM_SHARE` flags are cleared) belonging to two different processes in the same page frame; in this case, an exception should be generated when either one of the processes tries to modify the page.

- As mentioned in Chapter 2, 80×86 processors's Page Tables have just two protection bits, namely the `Read/Write` and `User/Supervisor` flags. Moreover, the `User/Supervisor` flag of every page included in a memory region must always be set, because the page must always be accessible by User Mode processes.

- Recent Intel Pentium 4 microprocessors with PAE enabled sport a `NX` (No eXecute) flag in each 64-bit Page Table entry.

If the kernel has been compiled without support for PAE, Linux adopts the following rules, which overcome the hardware limitation of the 80×86 microprocessors:

- The Read access right always implies the Execute access right, and vice versa.
- The Write access right always implies the Read access right.

Conversely, if the kernel has been compiled with support for PAE and the CPU has the `NX` flag, Linux adopts different rules:

- The Execute access right always implies the Read access right.
- The Write access right always implies the Read access right.

Moreover, to correctly defer the allocation of page frames through the "Copy On Write" technique (see later in this chapter), the page frame is write-protected whenever the corresponding page must not be shared by several processes.

Therefore, the 16 possible combinations of the Read, Write, Execute, and Share access rights are scaled down according to the following rules:

- If the page has both Write and Share access rights, the `Read/Write` bit is set.

- If the page has the Read or Execute access right but does not have either the Write or the Share access right, the `Read/Write` bit is cleared.

- If the `NX` bit is supported and the page does not have the Execute access right, the `NX` bit is set.

- If the page does not have any access rights, the `Present` bit is cleared so that each access generates a Page Fault exception. However, to distinguish this condition from the real page-not-present case, Linux also sets the `Page size` bit to 1.*

The downscaled protection bits corresponding to each combination of access rights are stored in the 16 elements of the `protection_map` array.

Memory Region Handling

Having the basic understanding of data structures and state information that control memory handling, we can look at a group of low-level functions that operate on memory region descriptors. They should be considered auxiliary functions that simplify the implementation of `do_mmap( )` and `do_munmap( )`. Those two functions, which are described in the sections "Allocating a Linear Address Interval" and "Releasing a Linear Address Interval" later in this chapter, enlarge and shrink the address space of a process, respectively. Working at a higher level than the functions we consider here, they do not receive a memory region descriptor as their parameter, but rather the initial address, the length, and the access rights of a linear address interval.

Finding the closest region to a given address: find_vma( )

The `find_vma( )` function acts on two parameters: the address `mm` of a process memory descriptor and a linear address `addr`. It locates the first memory region whose `vm_end` field is greater than `addr` and returns the address of its descriptor; if no such region exists, it returns a `NULL` pointer. Notice that the region selected by `find_vma( )` does not necessarily include `addr` because `addr` may lie outside of any memory region.

Each memory descriptor includes an `mmap_cache` field that stores the descriptor address of the region that was last referenced by the process. This additional field is introduced to reduce the time spent in looking for the region that contains a given linear address. Locality of address references in programs makes it highly likely that if the last linear address checked belonged to a given region, the next one to be checked belongs to the same region.

---

* You might consider this use of the `Page size` bit to be a dirty trick, because the bit was meant to indicate the real page size. But Linux can get away with the deception because the 80×86 chip checks the `Page size` bit in Page Directory entries, but not in Page Table entries.

The function thus starts by checking whether the region identified by mmap_cache includes addr. If so, it returns the region descriptor pointer:

```
vma = mm->mmap_cache;
if (vma && vma->vm_end > addr && vma->vm_start <= addr)
    return vma;
```

Otherwise, the memory regions of the process must be scanned, and the function looks up the memory region in the red-black tree:

```
rb_node = mm->mm_rb.rb_node;
vma = NULL;
while (rb_node) {
    vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            break;
        rb_node = rb_node->rb_left;
    } else
        rb_node = rb_node->rb_right;
}
if (vma)
    mm->mmap_cache = vma;
return vma;
```

The function uses the rb_entry macro, which derives from a pointer to a node of the red-black tree the address of the corresponding memory region descriptor.

The find_vma_prev( ) function is similar to find_vma( ), except that it writes in an additional pprev parameter a pointer to the descriptor of the memory region that precedes the one selected by the function.

Finally, the find_vma_prepare( ) function locates the position of the new leaf in the red-black tree that corresponds to a given linear address and returns the addresses of the preceding memory region and of the parent node of the leaf to be inserted.

Finding a region that overlaps a given interval: find_vma_intersection( )

The find_vma_intersection( ) function finds the first memory region that overlaps a given linear address interval; the mm parameter points to the memory descriptor of the process, while the start_addr and end_addr linear addresses specify the interval:

```
vma = find_vma(mm,start_addr);
if (vma && end_addr <= vma->vm_start)
    vma = NULL;
return vma;
```

The function returns a NULL pointer if no such region exists. To be exact, if find_vma( ) returns a valid address but the memory region found starts after the end of the linear address interval, vma is set to NULL.

Finding a free interval: get_unmapped_area( )

The `get_unmapped_area( )` function searches the process address space to find an available linear address interval. The `len` parameter specifies the interval length, while a non-null `addr` parameter specifies the address from which the search must be started. If the search is successful, the function returns the initial address of the new interval; otherwise, it returns the error code `-ENOMEM`.

If the `addr` parameter is not `NULL`, the function checks that the specified address is in the User Mode address space and that it is aligned to a page boundary. Next, the function invokes either one of two methods, depending on whether the linear address interval should be used for a file memory mapping or for an anonymous memory mapping. In the former case, the function executes the `get_unmapped_area` file operation; this is discussed in Chapter 16.

In the latter case, the function executes the `get_unmapped_area` method of the memory descriptor. In turn, this method is implemented by either the `arch_get_unmapped_area( )` function, or the `arch_get_unmapped_area_topdown( )` function, according to the memory region layout of the process. As we'll see in the section "Program Segments and Process Memory Regions" in Chapter 20, every process can have two different layouts for the memory regions allocated through the `mmap( )` system call: either they start from the linear address `0x40000000` and grow towards higher addresses, or they start right above the User Mode stack and grow towards lower addresses.

Let us discuss the `arch_get_unmapped_area( )` function, which is used when the memory regions are allocated moving from lower addresses to higher ones. It is essentially equivalent to the following code fragment:

```
if (len > TASK_SIZE)
    return -ENOMEM;
addr = (addr + 0xfff) & 0xfffff000;
if (addr && addr + len <= TASK_SIZE) {
    vma = find_vma(current->mm, addr);
    if (!vma || addr + len <= vma->vm_start)
        return addr;
}
start_addr = addr = mm->free_area_cache;
for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
    if (addr + len > TASK_SIZE) {
        if (start_addr == (TASK_SIZE/3+0xfff)&0xfffff000)
            return -ENOMEM;
        start_addr = addr = (TASK_SIZE/3+0xfff)&0xfffff000;
        vma = find_vma(current->mm, addr);
    }
    if (!vma || addr + len <= vma->vm_start) {
        mm->free_area_cache = addr + len;
        return addr;
    }
    addr = vma->vm_end;
}
```

The function starts by checking to make sure the interval length is within TASK_SIZE, the limit imposed on User Mode linear addresses (usually 3 GB). If addr is different from zero, the function tries to allocate the interval starting from addr. To be on the safe side, the function rounds up the value of addr to a multiple of 4 KB.

If addr is 0 or the previous search failed, the arch_get_unmapped_area() function scans the User Mode linear address space looking for a range of linear addresses not included in any memory region and large enough to contain the new region. To speed up the search, the search's starting point is usually set to the linear address following the last allocated memory region. The mm->free_area_cache field of the memory descriptor is initialized to one-third of the User Mode linear address space—usually, 1 GB—and then updated as new memory regions are created. If the function fails in finding a suitable range of linear addresses, the search restarts from the beginning—that is, from one-third of the User Mode linear address space: in fact, the first third of the User Mode linear address space is reserved for memory regions having a predefined starting linear address, typically the text, data, and bss segments of an executable file (see Chapter 20).

The function invokes find_vma() to locate the first memory region ending after the search's starting point, then repeatedly considers all the following memory regions. Three cases may occur:

- The requested interval is larger than the portion of linear address space yet to be scanned (addr + len > TASK_SIZE): in this case, the function either restarts from one-third of the User Mode address space or, if the second search has already been done, returns -ENOMEM (there are not enough linear addresses to satisfy the request).

- The hole following the last scanned region is not large enough (vma != NULL && vma->vm_start < addr + len). In this case, the function considers the next region.

- If neither one of the preceding conditions holds, a large enough hole has been found. In this case, the function returns addr.

Inserting a region in the memory descriptor list: insert_vm_struct( )

insert_vm_struct( ) inserts a vm_area_struct structure in the memory region object list and red-black tree of a memory descriptor. It uses two parameters: mm, which specifies the address of a process memory descriptor, and vma, which specifies the address of the vm_area_struct object to be inserted. The vm_start and vm_end fields of the memory region object must have already been initialized. The function invokes the find_vma_prepare() function to look up the position in the red-black tree mm->mm_rb where vma should go. Then insert_vm_struct( ) invokes the vma_link() function, which in turn:

1. Inserts the memory region in the linked list referenced by mm->mmap.

2. Inserts the memory region in the red-black tree `mm->mm_rb`.

3. If the memory region is anonymous, inserts the region in the list headed at the corresponding anon_vma data structure (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).

4. Increases the `mm->map_count` counter.

If the region contains a memory-mapped file, the `vma_link()` function performs additional tasks that are described in Chapter 17.

The `__vma_unlink()` function receives as its parameters a memory descriptor address `mm` and two memory region object addresses `vma` and `prev`. Both memory regions should belong to `mm`, and `prev` should precede `vma` in the memory region ordering. The function removes `vma` from the linked list and the red-black tree of the memory descriptor. It also updates `mm->mmap_cache`, which stores the last referenced memory region, if this field points to the memory region just deleted.

Allocating a Linear Address Interval

Now let's discuss how new linear address intervals are allocated. To do this, the `do_mmap()` function creates and initializes a new memory region for the `current` process. However, after a successful allocation, the memory region could be merged with other memory regions defined for the process.

The function uses the following parameters:

file *and* offset

File object pointer `file` and file offset `offset` are used if the new memory region will map a file into memory. This topic is discussed in Chapter 16. In this section, we assume that no memory mapping is required and that `file` and `offset` are both NULL.

addr

This linear address specifies where the search for a free interval must start.

len

The length of the linear address interval.

prot

This parameter specifies the access rights of the pages included in the memory region. Possible flags are PROT_READ, PROT_WRITE, PROT_EXEC, and PROT_NONE. The first three flags mean the same things as the VM_READ, VM_WRITE, and VM_EXEC flags. PROT_NONE indicates that the process has none of those access rights.

flag

This parameter specifies the remaining memory region flags:

MAP_GROWSDOWN, MAP_LOCKED, MAP_DENYWRITE, *and* MAP_EXECUTABLE

Their meanings are identical to those of the flags listed in Table 9-5.

MAP_SHARED *and* MAP_PRIVATE

> The former flag specifies that the pages in the memory region can be shared among several processes; the latter flag has the opposite effect. Both flags refer to the VM_SHARED flag in the vm_area_struct descriptor.

MAP_FIXED

> The initial linear address of the interval must be exactly the one specified in the addr parameter.

MAP_ANONYMOUS

> No file is associated with the memory region (see Chapter 16).

MAP_NORESERVE

> The function doesn't have to do a preliminary check on the number of free page frames.

MAP_POPULATE

> The function should pre-allocate the page frames required for the mapping established by the memory region. This flag is significant only for memory regions that map files (see Chapter 16) and for IPC shared memory regions (see Chapter 19).

MAP_NONBLOCK

> Significant only when the MAP_POPULATE flag is set: when pre-allocating the page frames, the function must not block.

The do_mmap( ) function performs some preliminary checks on the value of offset and then executes the do_mmap_pgoff( ) function. In this chapter we will suppose that the new interval of linear address does not map a file on disk—file memory mapping is discussed in detail in Chapter 16. Here is a description of the do_mmap_pgoff( ) function for anonymous memory regions:

1. Checks whether the parameter values are correct and whether the request can be satisfied. In particular, it checks for the following conditions that prevent it from satisfying the request:

   • The linear address interval has zero length or includes addresses greater than TASK_SIZE.

   • The process has already mapped too many memory regions—that is, the value of the map_count field of its mm memory descriptor exceeds the allowed maximum value.

   • The flag parameter specifies that the pages of the new linear address interval must be locked in RAM, but the process is not allowed to create locked memory regions, or the number of pages locked by the process exceeds the threshold stored in the signal->rlim[RLIMIT_MEMLOCK].rlim_cur field of the process descriptor.

   If any of the preceding conditions holds, do_mmap_pgoff( ) terminates by returning a negative value. If the linear address interval has a zero length, the function returns without performing any action.

2. Invokes `get_unmapped_area()` to obtain a linear address interval for the new region (see the previous section "Memory Region Handling").

3. Computes the flags of the new memory region by combining the values stored in the `prot` and `flags` parameters:

```
vm_flags = calc_vm_prot_bits(prot,flags) |
           calc_vm_flag_bits(prot,flags) |
           mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
if (flags & MAP_SHARED)
    vm_flags |= VM_SHARED | VM_MAYSHARE;
```

The `calc_vm_prot_bits()` function sets the `VM_READ`, `VM_WRITE`, and `VM_EXEC` flags in `vm_flags` only if the corresponding `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC` flags in `prot` are set. The `calc_vm_flag_bits()` function sets the `VM_GROWSDOWN`, `VM_DENYWRITE`, `VM_EXECUTABLE`, and `VM_LOCKED` flags in `vm_flags` only if the corresponding `MAP_GROWSDOWN`, `MAP_DENYWRITE`, `MAP_EXECUTABLE`, and `MAP_LOCKED` flags in `flags` are set. A few other flags are set in `vm_flags`: `VM_MAYREAD`, `VM_MAYWRITE`, `VM_MAYEXEC`, the default flags for all memory regions in `mm->def_flags`,[*] and both `VM_SHARED` and `VM_MAYSHARE` if the pages of the memory region have to be shared with other processes.

4. Invokes `find_vma_prepare()` to locate the object of the memory region that shall precede the new interval, as well as the position of the new region in the red-black tree:

```
for (;;) {
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
    if (!vma || vma->vm_start >= addr + len)
        break;
    if (do_munmap(mm, addr, len))
        return -ENOMEM;
}
```

The `find_vma_prepare()` function also checks whether a memory region that overlaps the new interval already exists. This occurs when the function returns a non-`NULL` address pointing to a region that starts before the end of the new interval. In this case, `do_mmap_pgoff()` invokes `do_munmap()` to remove the new interval and then repeats the whole step (see the later section "Releasing a Linear Address Interval").

5. Checks whether inserting the new memory region causes the size of the process address space (`mm->total_vm<<PAGE_SHIFT)+len` to exceed the threshold stored in the `signal->rlim[RLIMIT_AS].rlim_cur` field of the process descriptor. If so, it returns the error code `-ENOMEM`. Notice that the check is done here and not in step 1 with the other checks, because some memory regions could have been removed in step 4.

---

[*] Actually, the `def_flags` field of the memory descriptor is modified only by the `mlockall()` system call, which can be used to set the `VM_LOCKED` flag, thus locking all future pages of the calling process in RAM.

6. Returns the error code `-ENOMEM` if the `MAP_NORESERVE` flag was not set in the `flags` parameter, the new memory region contains private writable pages, and there are not enough free page frames; this last check is performed by the `security_vm_enough_memory()` function.

7. If the new interval is private (`VM_SHARED` not set) and it does not map a file on disk, it invokes `vma_merge()` to check whether the preceding memory region can be expanded in such a way to include the new interval. Of course, the preceding memory region must have exactly the same flags as those memory regions stored in the `vm_flags` local variable. If the preceding memory region can be expanded, `vma_merge()` also tries to merge it with the following memory region (this occurs when the new interval fills the hole between two memory regions and all three have the same flags). In case it succeeds in expanding the preceding memory region, the function jumps to step 12.

8. Allocates a `vm_area_struct` data structure for the new memory region by invoking the `kmem_cache_alloc()` slab allocator function.

9. Initializes the new memory region object (pointed to by `vma`):

```
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = protection_map[vm_flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = pgoff;
vma->vm_file = NULL;
vma->vm_private_data = NULL;
vma->vm_next = NULL;
INIT_LIST_HEAD(&vma->shared);
```

10. If the `MAP_SHARED` flag is set (and the new memory region doesn't map a file on disk), the region is a shared anonymous region: invokes `shmem_zero_setup()` to initialize it. Shared anonymous regions are mainly used for interprocess communications; see the section "IPC Shared Memory" in Chapter 19.

11. Invokes `vma_link()` to insert the new region in the memory region list and red-black tree (see the earlier section "Memory Region Handling").

12. Increases the size of the process address space stored in the `total_vm` field of the memory descriptor.

13. If the `VM_LOCKED` flag is set, it invokes `make_pages_present()` to allocate all the pages of the memory region in succession and lock them in RAM:

```
if (vm_flags & VM_LOCKED) {
    mm->locked_vm += len >> PAGE_SHIFT;
    make_pages_present(addr, addr + len);
}
```

The `make_pages_present()` function, in turn, invokes `get_user_pages()` as follows:

```
write = (vma->vm_flags & VM_WRITE) != 0;
get_user_pages(current, current->mm, addr, len, write, 0, NULL, NULL);
```

The get_user_pages() function cycles through all starting linear addresses of the pages between addr and addr+len; for each of them, it invokes follow_page() to check whether there is a mapping to a physical page in the current's Page Tables. If no such physical page exists, get_user_pages() invokes handle_mm_fault(), which, as we'll see in the section "Handling a Faulty Address Inside the Address Space," allocates one page frame and sets its Page Table entry according to the vm_flags field of the memory region descriptor.

14. Finally, it terminates by returning the linear address of the new memory region.

Releasing a Linear Address Interval

When the kernel must delete a linear address interval from the address space of the current process, it uses the do_munmap() function. The parameters are: the address mm of the process's memory descriptor, the starting address start of the interval, and its length len. The interval to be deleted does not usually correspond to a memory region; it may be included in one memory region or span two or more regions.

The do_munmap() function

The function goes through two main phases. In the first phase (steps 1–6), it scans the list of memory regions owned by the process and unlinks all regions included in the linear address interval from the process address space. In the second phase (steps 7–12), the function updates the process Page Tables and removes the memory regions identified in the first phase. The function makes use of the split_vma() and unmap_region() functions, which will be described later. do_munmap() executes the following steps:

1. Performs some preliminary checks on the parameter values. If the linear address interval includes addresses greater than TASK_SIZE, if start is not a multiple of 4,096, or if the linear address interval has a zero length, the function returns the error code -EINVAL.

2. Locates the first memory region mpnt that ends after the linear address interval to be deleted (mpnt->end > start), if any:

    ```
    mpnt = find_vma_prev(mm, start, &prev);
    ```

3. If there is no such memory region, or if the region does not overlap with the linear address interval, nothing has to be done because there is no memory region in the interval:

    ```
    end = start + len;
    if (!mpnt || mpnt->vm_start >= end)
        return 0;
    ```

4. If the linear address interval starts inside the mpnt memory region, it invokes split_vma() (described below) to split the mpnt memory region into two smaller regions: one outside the interval and the other inside the interval:

    ```
    if (start > mpnt->vm_start) {
        if (split_vma(mm, mpnt, start, 0))
    ```

```
        return -ENOMEM;
    prev = mpnt;
}
```

The prev local variable, which previously stored the pointer to the memory region preceding mpnt, is updated so that it points to mpnt—that is, to the new memory region lying outside the linear address interval. In this way, prev still points to the memory region preceding the first memory region to be removed.

5. If the linear address interval ends inside a memory region, it invokes split_vma() once again to split the last overlapping memory region into two smaller regions: one inside the interval and the other outside the interval:[*]

```
last = find_vma(mm, end);
if (last && end > last->vm_start)){
    if (split_vma(mm, last, start, end, 1))
        return -ENOMEM;
}
```

6. Updates the value of mpnt so that it points to the first memory region in the linear address interval. If prev is NULL—that is, there is no preceding memory region—the address of the first memory region is taken from mm->mmap:

```
mpnt = prev ? prev->vm_next : mm->mmap;
```

7. Invokes detach_vmas_to_be_unmapped()to remove the memory regions included in the linear address interval from the process's linear address space. This function essentially executes the following code:

```
vma = mpnt;
insertion_point = (prev ? &prev->vm_next : &mm->mmap);
do {
    rb_erase(&vma->vm_rb, &mm->mm_rb);
    mm->map_count--;
    tail_vma = vma;
    vma = vma->next;
} while (vma && vma->start < end);
*insertion_point = vma;
tail_vma->vm_next = NULL;
mm->map_cache = NULL;
```

The descriptors of the regions to be removed are stored in an ordered list, whose head is pointed to by the mpnt local variable (actually, this list is just a fragment of the original process's list of memory regions).

8. Gets the mm->page_table_lock spin lock.

9. Invokes unmap_region() to clear the Page Table entries covering the linear address interval and to free the corresponding page frames (discussed later):

```
unmap_region(mm, mpnt, prev, start, end);
```

10. Releases the mm->page_table_lock spin lock.

---

[*] If the linear address interval is properly contained inside a memory region, the region must be replaced by two new smaller regions. When this case occurs, step 4 and step 5 break the memory region in three smaller regions: the middle region is destroyed, while the first and the last ones will be preserved.

11. Releases the descriptors of the memory regions collected in the list built in step 7:

```
do {
    struct vm_area_struct * next = mpnt->vm_next;
    unmap_vma(mm, mpnt);
    mpnt = next;
} while (mpnt != NULL);
```

The `unmap_vma()` function is invoked on every memory region in the list; it essentially executes the following steps:

a. Updates the `mm->total_vm` and `mm->locked_vm` fields.

b. Executes the `mm->unmap_area` method of the memory descriptor. This method is implemented either by `arch_unmap_area()` or by `arch_unmap_area_topdown()`, according to the memory region layout of the process (see the earlier section "Memory Region Handling"). In both cases, the `mm->free_area_cache` field is updated, if needed.

c. Invokes the `close` method of the memory region, if defined.

d. If the memory region is anonymous, the function removes it from the anonymous memory region list headed at `mm->anon_vma`.

e. Invokes `kmem_cache_free()` to release the memory region descriptor.

12. Returns 0 (success).

The split_vma() function

The purpose of the `split_vma()` function is to split a memory region that intersects a linear address interval into two smaller regions, one outside of the interval and the other inside. The function receives four parameters: a memory descriptor pointer `mm`, a memory area descriptor pointer `vma` that identifies the region to be split, an address `addr` that specifies the intersection point between the interval and the memory region, and a flag `new_below` that specifies whether the intersection occurs at the beginning or at the end of the interval. The function performs the following basic steps:

1. Invokes `kmem_cache_alloc()` to get an additional `vm_area_struct` descriptor, and stores its address in the `new` local variable. If no free memory is available, it returns `-ENOMEM`.

2. Initializes the fields of the `new` descriptor with the contents of the fields of the `vma` descriptor.

3. If the `new_below` flag is 0, the linear address interval starts inside the `vma` region, so the new region must be placed after the `vma` region. Thus, the function sets both the `new->vm_start` and the `vma->vm_end` fields to `addr`.

4. Conversely, if the `new_below` flag is equal to 1, the linear address interval ends inside the `vma` region, so the new region must be placed before the `vma` region. Thus, the function sets both the `new->vm_end` and the `vma->vm_start` fields to `addr`.

5. If the open method of the new memory region is defined, the function executes it.

6. Links the new memory region descriptor to the mm->mmap list of memory regions and to the mm->mm_rb red-black tree. Moreover, the function adjusts the red-black tree to take care of the new size of the memory region vma.

7. Returns 0 (success).

The unmap_region( ) function

The unmap_region( ) function walks through a list of memory regions and releases the page frames belonging to them. It acts on five parameters: a memory descriptor pointer mm, a pointer vma to the descriptor of the first memory region being removed, a pointer prev to the memory region preceding vma in the process's list (see steps 2 and 4 in do_munmap()), and two addresses start and end that delimit the linear address interval being removed. The function essentially executes the following steps:

1. Invokes lru_add_drain( ) (see Chapter 17).

2. Invokes the tlb_gather_mmu( ) function to initialize a per-CPU variable named mmu_gathers. The contents of mmu_gathers are architecture-dependent: generally speaking, the variable should store all information required for a successful updating of the page table entries of a process. In the 80×86 architecture, the tlb_gather_mmu( ) function simply saves the value of the mm memory descriptor pointer in the mmu_gathers variable of the local CPU.

3. Stores the address of the mmu_gathers variable in the tlb local variable.

4. Invokes unmap_vmas( ) to scan all Page Table entries belonging to the linear address interval: if only one CPU is available, the function invokes free_swap_ and_cache( ) repeatedly to release the corresponding pages (see Chapter 17); otherwise, the function saves the pointers of the corresponding page descriptors in the mmu_gathers local variable.

5. Invokes free_pgtables(tlb,prev,start,end) to try to reclaim the Page Tables of the process that have been emptied in the previous step.

6. Invokes tlb_finish_mmu(tlb,start,end) to finish the work: in turn, this function:

   a. Invokes flush_tlb_mm( ) to flush the TLB (see the section "Handling the Hardware Cache and the TLB" in Chapter 2).

   b. In multiprocessor system, invokes free_pages_and_swap_cache( ) to release the page frames whose pointers have been collected in the mmu_gather data structure. This function is described in Chapter 17.

Page Fault Exception Handler

As stated previously, the Linux Page Fault exception handler must distinguish exceptions caused by programming errors from those caused by a reference to a page that legitimately belongs to the process address space but simply hasn't been allocated yet.

The memory region descriptors allow the exception handler to perform its job quite efficiently. The do_page_fault( ) function, which is the Page Fault interrupt service routine for the 80×86 architecture, compares the linear address that caused the Page Fault against the memory regions of the current process; it can thus determine the proper way to handle the exception according to the scheme that is illustrated in Figure 9-4.
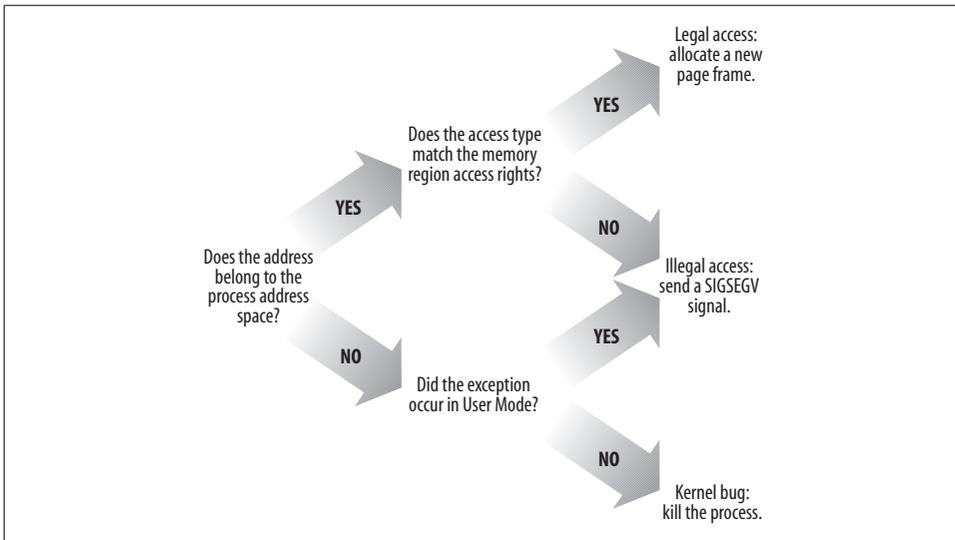
Figure 9-4. Overall scheme for the Page Fault handler

In practice, things are a lot more complex because the Page Fault handler must recognize several particular subcases that fit awkwardly into the overall scheme, and it must distinguish several kinds of legal access. A detailed flow diagram of the handler is illustrated in Figure 9-5.

The identifiers vmalloc_fault, good_area, bad_area, and no_context are labels appearing in do_page_fault( ) that should help you to relate the blocks of the flow diagram to specific lines of code.

The do_page_fault( ) function accepts the following input parameters:

- The regs address of a pt_regs structure containing the values of the microprocessor registers when the exception occurred.
- A 3-bit error_code, which is pushed on the stack by the control unit when the exception occurred (see "Hardware Handling of Interrupts and Exceptions" in Chapter 4). The bits have the following meanings:
  — If bit 0 is clear, the exception was caused by an access to a page that is not present (the Present flag in the Page Table entry is clear); otherwise, if bit 0 is set, the exception was caused by an invalid access right.

Access to kernel space — YES / NO

YES — Access in Kernel Mode — NO

vmalloc_fault

YES — Noncontiguous memory area address — NO

In interrupt, softirq, critical region, or kernel thread — NO / YES

Kernel page table entry fixup

YES — Address in a memory region — NO

good_area

YES — Address could belong to User Mode stack — NO

bad_area

YES — Write access — NO

YES — Access in User Mode — NO

NO — Page is present — YES

no_context

YES — Region is writable — NO

NO — Address is a wrong system call parameter — YES

YES — Region is readable or executable — NO

do_sigbus

Demand Paging and/or Copy On Write

Demand paging

Send SIGSEGV

Kill process and kernel "Oops"

"Fixup code" (typically send SIGSEGV)

Figure 9-5. The flow diagram of the Page Fault handler

— If bit 1 is clear, the exception was caused by a read or execute access; if set, the exception was caused by a write access.

— If bit 2 is clear, the exception occurred while the processor was in Kernel Mode; otherwise, it occurred in User Mode.

The first operation of do_page_fault( ) consists of reading the linear address that caused the Page Fault. When the exception occurs, the CPU control unit stores that value in the cr2 control register:

```
asm("movl %%cr2,%0":"=r" (address));
if (regs->eflags & 0x00020200)
```

```
        local_irq_enable();
    tsk = current;
```

The linear address is saved in the address local variable. The function also ensures that local interrupts are enabled if they were enabled before the fault or the CPU was running in virtual-8086 mode, and saves the pointers to the process descriptor of current in the tsk local variable.

As shown at the top of Figure 9-5, do_page_fault( ) checks whether the faulty linear address belongs to the fourth gigabyte:

```
info.si_code = SEGV_MAPERR;
if (address >= TASK_SIZE ) {
    if  (!(error_code & 0x101))
        goto vmalloc_fault;
    goto bad_area_nosemaphore;
}
```

If the exception was caused by the kernel trying to access a nonexisting page frame, a jump is made to the code at label vmalloc_fault, which takes care of faults that were likely caused by accessing a noncontiguous memory area in Kernel Mode; we describe this case in the later section "Handling Noncontiguous Memory Area Accesses." Otherwise, a jump is made to the code at the bad_area_nosemaphore label, described in the later section "Handling a Faulty Address Outside the Address Space."

Next, the handler checks whether the exception occurred while the kernel was executing some critical routine or running a kernel thread (remember that the mm field of the process descriptor is always NULL for kernel threads):

```
if (in_atomic() || !tsk->mm)
    goto bad_area_nosemaphore;
```

The in_atomic() macro yields the value one if the fault occurred while either one of the following conditions holds:

- The kernel was executing an interrupt handler or a deferrable function.
- The kernel was executing a critical region with kernel preemption disabled (see the section "Kernel Preemption" in Chapter 5).

If the Page Fault did occur in an interrupt handler, in a deferrable function, in a critical region, or in a kernel thread, do_page_fault( ) does not try to compare the linear address with the memory regions of current. Kernel threads never use linear addresses below TASK_SIZE. Similarly, interrupt handlers, deferrable functions, and code of critical regions should not use linear addresses below TASK_SIZE because this might block the current process. (See the section "Handling a Faulty Address Outside the Address Space" later in this chapter for information on the info local variable and a description of the code at the bad_area_nosemaphore label.)

Let's suppose that the Page Fault did not occur in an interrupt handler, in a deferrable function, in a critical region, or in a kernel thread. Then the function must inspect the memory regions owned by the process to determine whether the faulty

linear address is included in the process address space. In order to this, it must acquire the `mmap_sem` read/write semaphore of the process:

```
if (!down_read_trylock(&tsk->mm>mmap_sem)) {
    if ((error_code & 4) == 0 &&
        !search_exception_table(regs->eip))
        goto bad_area_nosemaphore;
    down_read(&tsk->mm->mmap_sem);
}
```

If kernel bugs and hardware malfunctioning can be ruled out, the current process has not already acquired the `mmap_sem` semaphore for writing when the Page Fault occurs. However, `do_page_fault()` wants to be sure that this is actually true, because otherwise a deadlock would occur. For that reason, the function makes use of `down_read_trylock()` instead of `down_read()` (see the section "Read/Write Semaphores" in Chapter 5). If the semaphore is closed and the Page Fault occurred in Kernel Mode, `do_page_fault()` determines whether the exception occurred while using some linear address that has been passed to the kernel as a parameter of a system call (see the next section "Handling a Faulty Address Outside the Address Space"). In this case, `do_page_fault()` knows for sure that the semaphore is owned by another process— because every system call service routine carefully avoids acquiring the `mmap_sem` semaphore for writing before accessing the User Mode address space—so the function waits until the semaphore is released. Otherwise, the Page Fault is due to a kernel bug or to a serious hardware problem, so the function jumps to the `bad_area_nosemaphore` label.

Let's assume that the `mmap_sem` semaphore has been safely acquired for reading. Now `do_page_fault()` looks for a memory region containing the faulty linear address:

```
vma = find_vma(tsk->mm, address);
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
    goto good_area;
```

If `vma` is `NULL`, there is no memory region ending after `address`, and thus the faulty address is certainly bad. On the other hand, if the first memory region ending after `address` includes `address`, the function jumps to the code at label `good_area`.

If none of the two "if" conditions are satisfied, the function has determined that `address` is not included in any memory region; however, it must perform an additional check, because the faulty address may have been caused by a `push` or `pusha` instruction on the User Mode stack of the process.

Let's make a short digression to explain how stacks are mapped into memory regions. Each region that contains a stack expands toward lower addresses; its `VM_GROWSDOWN` flag is set, so the value of its `vm_end` field remains fixed while the value of its `vm_start` field may be decreased. The region boundaries include, but do not delimit precisely, the current size of the User Mode stack. The reasons for the fuzz factor are:

- The region size is a multiple of 4 KB (it must include complete pages) while the stack size is arbitrary.
- Page frames assigned to a region are never released until the region is deleted; in particular, the value of the vm_start field of a region that includes a stack can only decrease; it can never increase. Even if the process executes a series of pop instructions, the region size remains unchanged.

It should now be clear how a process that has filled up the last page frame allocated to its stack may cause a Page Fault exception: the push refers to an address outside of the region (and to a nonexistent page frame). Notice that this kind of exception is not caused by a programming error; thus it must be handled separately by the Page Fault handler.

We now return to the description of do_page_fault( ), which checks for the case described previously:

```
if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;
if (error_code & 4      /* User Mode */
    && address + 32 < regs->esp)
        goto bad_area;
if (expand_stack(vma, address))
    goto bad_area;
goto good_area;
```

If the VM_GROWSDOWN flag of the region is set and the exception occurred in User Mode, the function checks whether address is smaller than the regs->esp stack pointer (it should be only a little smaller). Because a few stack-related assembly language instructions (such as pusha) perform a decrement of the esp register only after the memory access, a 32-byte tolerance interval is granted to the process. If the address is high enough (within the tolerance granted), the code invokes the expand_stack( ) function to check whether the process is allowed to extend both its stack and its address space; if everything is OK, it sets the vm_start field of vma to address and returns 0; otherwise, it returns -ENOMEM.

Note that the preceding code skips the tolerance check whenever the VM_GROWSDOWN flag of the region is set and the exception did not occur in User Mode. These conditions mean that the kernel is addressing the User Mode stack and that the code should always run expand_stack( ).

Handling a Faulty Address Outside the Address Space

If address does not belong to the process address space, do_page_fault( ) proceeds to execute the statements at the label bad_area. If the error occurred in User Mode, it sends a SIGSEGV signal to current (see the section "Generating a Signal" in Chapter 11) and terminates:

```
bad_area:
up_read(&tsk->mm->mmap_sem);
```

```
bad_area_nosemaphore:
if (error_code & 4) {    /* User Mode */
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code | (address >= TASK_SIZE);
    tsk->thread.trap_no = 14;
    info.si_signo = SIGSEGV;
    info.si_errno = 0;
    info.si_addr = (void *) address;
    force_sig_info(SIGSEGV, &info, tsk);
    return;
}
```

The force_sig_info() function makes sure that the process does not ignore or block the SIGSEGV signal, and sends the signal to the User Mode process while passing some additional information in the info local variable (see the section "Generating a Signal" in Chapter 11). The info.si_code field is already set to SEGV_MAPERR (if the exception was due to a nonexisting page frame) or to SEGV_ACCERR (if the exception was due to an invalid access to an existing page frame).

If the exception occurred in Kernel Mode (bit 2 of error_code is clear), there are still two alternatives:

- The exception occurred while using some linear address that has been passed to the kernel as a parameter of a system call.
- The exception is due to a real kernel bug.

The function distinguishes these two alternatives as follows:

```
no_context:
if ((fixup = search_exception_table(regs->eip)) != 0) {
    regs->eip = fixup;
    return;
}
```

In the first case, it jumps to a "fixup code," which typically sends a SIGSEGV signal to current or terminates a system call handler with a proper error code (see the section "Dynamic Address Checking: The Fix-up Code" in Chapter 10).

In the second case, the function prints a complete dump of the CPU registers and of the Kernel Mode stack both on the console and on a system message buffer; it then kills the current process by invoking the do_exit() function (see Chapter 20). This is the so-called *"Kernel oops"* error, named after the message displayed. The dumped values can be used by kernel hackers to reconstruct the conditions that triggered the bug, and thus find and correct it.

Handling a Faulty Address Inside the Address Space

If address belongs to the process address space, do_page_fault() proceeds to the statement labeled good_area:

```
good_area:
info.si_code = SEGV_ACCERR;
```

```
        write = 0;
        if (error_code & 2) { /* write access */
            if (!(vma->vm_flags & VM_WRITE))
                goto bad_area;
            write++;
        } else                  /* read access */
            if ((error_code & 1) || !(vma->vm_flags & (VM_READ | VM_EXEC)))
                goto bad_area;
```

If the exception was caused by a write access, the function checks whether the memory region is writable. If not, it jumps to the bad_area code; if so, it sets the write local variable to 1.

If the exception was caused by a read or execute access, the function checks whether the page is already present in RAM. In this case, the exception occurred because the process tried to access a privileged page frame (one whose User/Supervisor flag is clear) in User Mode, so the function jumps to the bad_area code.[*] If the page is not present, the function also checks whether the memory region is readable or executable.

If the memory region access rights match the access type that caused the exception, the handle_mm_fault( ) function is invoked to allocate a new page frame:

```
    survive:
    ret = handle_mm_fault(tsk->mm, vma, address, write);
    if (ret == VM_FAULT_MINOR || ret == VM_FAULT_MAJOR) {
        if (ret == VM_FAULT_MINOR) tsk->min_flt++; else tsk->maj_flt++;
        up_read(&tsk->mm->mmap_sem);
        return;
    }
```

The handle_mm_fault( ) function returns VM_FAULT_MINOR or VM_FAULT_MAJOR if it succeeded in allocating a new page frame for the process. The value VM_FAULT_MINOR indicates that the Page Fault has been handled without blocking the current process; this kind of Page Fault is called *minor fault*. The value VM_FAULT_MAJOR indicates that the Page Fault forced the current process to sleep (most likely because time was spent while filling the page frame assigned to the process with data read from disk); a Page Fault that blocks the current process is called a *major fault*. The function can also return VM_FAULT_OOM (for not enough memory) or VM_FAULT_SIGBUS (for every other error).

If handle_mm_fault( ) returns the value VM_FAULT_SIGBUS, a SIGBUS signal is sent to the process:

```
    if (ret == VM_FAULT_SIGBUS) {
    do_sigbus:
        up_read(&tsk->mm->mmap_sem);
        if (!(error_code & 4)) /* Kernel Mode */
            goto no_context;
        tsk->thread.cr2 = address;
```

---

[*] However, this case should never happen, because the kernel does not assign privileged page frames to the processes.

```
        tsk->thread.error_code = error_code;
        tsk->thread.trap_no = 14;
        info.si_signo = SIGBUS;
        info.si_errno = 0;
        info.si_code = BUS_ADRERR;
        info.si_addr = (void *) address;
        force_sig_info(SIGBUS, &info, tsk);
    }
```

If `handle_mm_fault()` cannot allocate the new page frame, it returns the value `VM_FAULT_OOM`; in this case, the kernel usually kills the current process. However, if current is the *init* process, it is just put at the end of the run queue and the scheduler is invoked; once init resumes its execution, `handle_mm_fault()` is executed again:

```
if (ret == VM_FAULT_OOM) {
  out_of_memory:
    up_read(&tsk->mm->mmap_sem);
    if (tsk->pid != 1) {
        if (error_code & 4) /* User Mode */
            do_exit(SIGKILL);
        goto no_context;
    }
    yield();
    down_read(&tsk->mm->mmap_sem);
    goto survive;
}
```

The `handle_mm_fault()` function acts on four parameters:

mm

A pointer to the memory descriptor of the process that was running on the CPU when the exception occurred

vma

A pointer to the descriptor of the memory region, including the linear address that caused the exception

address

The linear address that caused the exception

write_access

Set to 1 if `tsk` attempted to write in `address` and to 0 if `tsk` attempted to read or execute it

The function starts by checking whether the Page Middle Directory and the Page Table used to map `address` exist. Even if `address` belongs to the process address space, the corresponding Page Tables might not have been allocated, so the task of allocating them precedes everything else:

```
pgd = pgd_offset(mm, address);
spin_lock(&mm->page_table_lock);
pud = pud_alloc(mm, pgd, address);
if (pud) {
    pmd = pmd_alloc(mm, pud, address);
```

```
        if (pmd) {
            pte = pte_alloc_map(mm, pmd, address);
            if (pte)
                return handle_pte_fault(mm, vma, address,
                                        write_access, pte, pmd);
        }
    }
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_OOM;
```

The `pgd` local variable contains the Page Global Directory entry that refers to `address`; `pud_alloc()` and `pmd_alloc()` are invoked to allocate, if needed, a new Page Upper Directory and a new Page Middle Directory, respectively.[*] `pte_alloc_map()` is then invoked to allocate, if needed, a new Page Table. If both operations are successful, the `pte` local variable points to the Page Table entry that refers to `address`. The `handle_pte_fault()` function is then invoked to inspect the Page Table entry corresponding to `address` and to determine how to allocate a new page frame for the process:

- If the accessed page is not present—that is, if it is not already stored in any page frame—the kernel allocates a new page frame and initializes it properly; this technique is called demand paging.

- If the accessed page is present but is marked read-only—i.e., if it is already stored in a page frame—the kernel allocates a new page frame and initializes its contents by copying the old page frame data; this technique is called *Copy On Write*.

Demand Paging

The term *demand paging* denotes a dynamic memory allocation technique that consists of deferring page frame allocation until the last possible moment—until the process attempts to address a page that is not present in RAM, thus causing a Page Fault exception.

The motivation behind demand paging is that processes do not address all the addresses included in their address space right from the start; in fact, some of these addresses may never be used by the process. Moreover, the program locality principle (see the section "Hardware Cache" in Chapter 2) ensures that, at each stage of program execution, only a small subset of the process pages are really referenced, and therefore the page frames containing the temporarily useless pages can be used by other processes. Demand paging is thus preferable to global allocation (assigning all page frames to the process right from the start and leaving them in memory until program termination), because it increases the average number of free page frames in the system and therefore allows better use of the available free memory. From

---

[*] On 80×86 microprocessors, these allocations never occur, because the Page Upper Directories are always included in the Page Global Directory, and the Page Middle Directories are either included in the Page Upper Directory (PAE not enabled) or allocated together with the Page Upper Directory (PAE enabled).

another viewpoint, it allows the system as a whole to get better throughput with the same amount of RAM.

The price to pay for all these good things is system overhead: each Page Fault exception induced by demand paging must be handled by the kernel, thus wasting CPU cycles. Fortunately, the locality principle ensures that once a process starts working with a group of pages, it sticks with them without addressing other pages for quite a while. Thus, Page Fault exceptions may be considered rare events.

An addressed page may not be present in main memory either because the page was never accessed by the process, or because the corresponding page frame has been reclaimed by the kernel (see Chapter 17).

In both cases, the page fault handler must assign a new page frame to the process. How this page frame is initialized, however, depends on the kind of page and on whether the page was previously accessed by the process. In particular:

1. Either the page was never accessed by the process and it does not map a disk file, or the page maps a disk file. The kernel can recognize these cases because the Page Table entry is filled with zeros—i.e., the `pte_none` macro returns the value 1.

2. The page belongs to a non-linear disk file mapping (see the section "Non-Linear Memory Mappings" in Chapter 16). The kernel can recognize this case, because the `Present` flag is cleared and the `Dirty` flag is set—i.e., the `pte_file` macro returns the value 1.

3. The page was already accessed by the process, but its content is temporarily saved on disk. The kernel can recognize this case because the Page Table entry is not filled with zeros, but the `Present` and `Dirty` flags are cleared.

Thus, the `handle_pte_fault()` function is able to distinguish the three cases by inspecting the Page Table entry that refers to `address`:

```
entry = *pte;
if (!pte_present(entry)) {
    if (pte_none(entry))
        return do_no_page(mm, vma, address, write_access, pte, pmd);
    if (pte_file(entry))
        return do_file_page(mm, vma, address, write_access, pte, pmd);
    return do_swap_page(mm, vma, address, pte, pmd, entry, write_access);
}
```

We'll examine cases 2 and 3 in Chapter 16 and in Chapter 17, respectively.

In case 1, when the page was never accessed or the page linearly maps a disk file, the `do_no_page()` function is invoked. There are two ways to load the missing page, depending on whether the page is mapped to a disk file. The function determines this by checking the `nopage` method of the `vma` memory region object, which points to the function that loads the missing page from disk into RAM if the page is mapped to a file. Therefore, the possibilities are:

- The vma->vm_ops->nopage field is not NULL. In this case, the memory region maps a disk file and the field points to the function that loads the page. This case is covered in the section "Demand Paging for Memory Mapping" in Chapter 16 and in the section "IPC Shared Memory" in Chapter 19.

- Either the vma->vm_ops field or the vma->vm_ops->nopage field is NULL. In this case, the memory region does not map a file on disk—i.e., it is an *anonymous mapping*. Thus, do_no_page( ) invokes the do_anonymous_page( ) function to get a new page frame:

```
if (!vma->vm_ops || !vma->vm_ops->nopage)
    return do_anonymous_page(mm, vma, page_table, pmd,
                              write_access, address);
```

The do_anonymous_page( ) function* handles write and read requests separately:

```
if (write_access) {
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    page = alloc_page(GFP_HIGHUSER | __GFP_ZERO);
    spin_lock(&mm->page_table_lock);
    page_table = pte_offset_map(pmd, addr);
    mm->rss++;
    entry = maybe_mkwrite(pte_mkdirty(mk_pte(page,
                                         vma->vm_page_prot)), vma);
    lru_cache_add_active(page);
    SetPageReferenced(page);
    set_pte(page_table, entry);
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
}
```

The first execution of the pte_unmap macro releases the temporary kernel mapping for the high-memory physical address of the Page Table entry established by pte_offset_map before invoking the handle_pte_fault() function (see Table 2-7 in the section "Page Table Handling" in Chapter 2). The following pair or pte_offset_map and pte_unmap macros acquires and releases the same temporary kernel mapping. The temporary kernel mapping has to be released before invoking alloc_page( ), because this function might block the current process.

The function increases the rss field of the memory descriptor to keep track of the number of page frames allocated to the process. The Page Table entry is then set to the physical address of the page frame, which is marked as writable† and dirty. The lru_cache_add_active( ) function inserts the new page frame in the swap-related data structures; we discuss it in Chapter 17.

---

* To simplify the description of this function, we skip the statements that deal with reverse mapping, a topic that will be covered in the section "Reverse Mapping" in Chapter 17.

† If a debugger attempts to write in a page belonging to a read-only memory region of the traced process, the kernel does not set the Read/Write flag. The maybe_mkwrite() function takes care of this special case.

Conversely, when handling a read access, the content of the page is irrelevant because the process is addressing it for the first time. It is safer to give a page filled with zeros to the process rather than an old page filled with information written by some other process. Linux goes one step further in the spirit of demand paging. There is no need to assign a new page frame filled with zeros to the process right away, because we might as well give it an existing page called *zero page*, thus deferring further page frame allocation. The zero page is allocated statically during kernel initialization in the `empty_zero_page` variable (an array of 4,096 bytes filled with zeros).

The Page Table entry is thus set with the physical address of the zero page:

```
entry = pte_wrprotect(mk_pte(virt_to_page(empty_zero_page),
                             vma->vm_page_prot));
set_pte(page_table, entry);
spin_unlock(&mm->page_table_lock);
return VM_FAULT_MINOR;
```

Because the page is marked as nonwritable, if the process attempts to write in it, the Copy On Write mechanism is activated. Only then does the process get a page of its own to write in. The mechanism is described in the next section.

Copy On Write

First-generation Unix systems implemented process creation in a rather clumsy way: when a fork( ) system call was issued, the kernel duplicated the whole parent address space in the literal sense of the word and assigned the copy to the child process. This activity was quite time consuming since it required:

- Allocating page frames for the Page Tables of the child process
- Allocating page frames for the pages of the child process
- Initializing the Page Tables of the child process
- Copying the pages of the parent process into the corresponding pages of the child process

This way of creating an address space involved many memory accesses, used up many CPU cycles, and completely spoiled the cache contents. Last but not least, it was often pointless because many child processes start their execution by loading a new program, thus discarding entirely the inherited address space (see Chapter 20).

Modern Unix kernels, including Linux, follow a more efficient approach called *Copy On Write* (*COW*). The idea is quite simple: instead of duplicating page frames, they are shared between the parent and the child process. However, as long as they are shared, they cannot be modified. Whenever the parent or the child process attempts to write into a shared page frame, an exception occurs. At this point, the kernel duplicates the page into a new page frame that it marks as writable. The original page frame remains write-protected: when the other process tries to write into it, the

kernel checks whether the writing process is the only owner of the page frame; in such a case, it makes the page frame writable for the process.

The _count field of the page descriptor is used to keep track of the number of processes that are sharing the corresponding page frame. Whenever a process releases a page frame or a Copy On Write is executed on it, its _count field is decreased; the page frame is freed only when _count becomes -1 (see the section "Page Descriptors" in Chapter 8).

Let's now describe how Linux implements COW. When handle_pte_fault() determines that the Page Fault exception was caused by an access to a page present in memory, it executes the following instructions:

```
if (pte_present(entry)) {
    if (write_access) {
        if (!pte_write(entry))
            return do_wp_page(mm, vma, address, pte, pmd, entry);
        entry = pte_mkdirty(entry);
    }
    entry = pte_mkyoung(entry);
    set_pte(pte, entry);
    flush_tlb_page(vma, address);
    pte_unmap(pte);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
}
```

The handle_pte_fault() function is architecture-independent: it considers each possible violation of the page access rights. However, in the 80 × 86 architecture, if the page is present, the access was for writing and the page frame is write-protected (see the earlier section "Handling a Faulty Address Inside the Address Space"). Thus, the do_wp_page() function is always invoked.

The do_wp_page() function* starts by deriving the page descriptor of the page frame referenced by the Page Table entry involved in the Page Fault exception. Next, the function determines whether the page must really be duplicated. If only one process owns the page, Copy On Write does not apply, and the process should be free to write the page. Basically, the function reads the _count field of the page descriptor: if it is equal to 0 (a single owner), COW must not be done. Actually, the check is slightly more complicated, because the _count field is also increased when the page is inserted into the swap cache (see the section "The Swap Cache" in Chapter 17) and when the PG_private flag in the page descriptor is set. However, when COW is not to be done, the page frame is marked as writable, so that it does not cause further Page Fault exceptions when writes are attempted:

```
set_pte(page_table, maybe_mkwrite(pte_mkyoung(pte_mkdirty(pte)),vma));
flush_tlb_page(vma, address);
```

---

* To simplify the description of this function, we skip the statements that deal with reverse mapping, a topic that will be covered in the section "Reverse Mapping" in Chapter 17.

```
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
```

If the page is shared among several processes by means of COW, the function copies the content of the old page frame (old_page) into the newly allocated one (new_page). To avoid race conditions, get_page() is invoked to increase the usage counter of old_page before starting the copy operation:

```
    old_page = pte_page(pte);
    pte_unmap(page_table);
    get_page(old_page);
    spin_unlock(&mm->page_table_lock);
    if (old_page == virt_to_page(empty_zero_page))
        new_page = alloc_page(GFP_HIGHUSER | __GFP_ZERO);
    } else {
        new_page = alloc_page(GFP_HIGHUSER);
        vfrom = kmap_atomic(old_page, KM_USER0)
        vto = kmap_atomic(new_page, KM_USER1);
        copy_page(vto, vfrom);
        kunmap_atomic(vfrom, KM_USER0);
        kunmap_atomic(vto, KM_USER0);
    }
```

If the old page is the zero page, the new frame is efficiently filled with zeros when it is allocated (__GFP_ZERO flag). Otherwise, the page frame content is copied using the copy_page() macro. Special handling for the zero page is not strictly required, but it improves the system performance, because it preserves the microprocessor hardware cache by making fewer address references.

Because the allocation of a page frame can block the process, the function checks whether the Page Table entry has been modified since the beginning of the function (pte and *page_table do not have the same value). In this case, the new page frame is released, the usage counter of old_page is decreased (to undo the increment made previously), and the function terminates.

If everything looks OK, the physical address of the new page frame is finally written into the Page Table entry, and the corresponding TLB register is invalidated:

```
    spin_lock(&mm->page_table_lock);
    entry = maybe_mkwrite(pte_mkdirty(mk_pte(new_page,
                                          vma->vm_page_prot)),vma);
    set_pte(page_table, entry);
    flush_tlb_page(vma, address);
    lru_cache_add_active(new_page);
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
```

The lru_cache_add_active() function inserts the new page frame in the swap-related data structures; see Chapter 17 for its description.

Finally, do_wp_page() decreases the usage counter of old_page twice. The first decrement undoes the safety increment made before copying the page frame contents; the

second decrement reflects the fact that the current process no longer owns the page frame.

Handling Noncontiguous Memory Area Accesses

We have seen in the section "Noncontiguous Memory Area Management" in Chapter 8 that the kernel is quite lazy in updating the Page Table entries corresponding to noncontiguous memory areas. In fact, the vmalloc() and vfree() functions limit themselves to updating the master kernel Page Tables (i.e., the Page Global Directory init_mm.pgd and its child Page Tables).

However, once the kernel initialization phase ends, the master kernel Page Tables are not directly used by any process or kernel thread. Thus, consider the first time that a process in Kernel Mode accesses a noncontiguous memory area. When translating the linear address into a physical address, the CPU's memory management unit encounters a null Page Table entry and raises a Page Fault. However, the handler recognizes this special case because the exception occurred in Kernel Mode, and the faulty linear address is greater than TASK_SIZE. Thus, the do_page_fault() handler checks the corresponding master kernel Page Table entry:

```
vmalloc_fault:
asm("movl %%cr3,%0":"=r" (pgd_paddr));
pgd = pgd_index(address) + (pgd_t *) __va(pgd_paddr);
pgd_k = init_mm.pgd + pgd_index(address);
if (!pgd_present(*pgd_k))
    goto no_context;
pud = pud_offset(pgd, address);
pud_k = pud_offset(pgd_k, address);
if (!pud_present(*pud_k))
    goto no_context;
pmd = pmd_offset(pud, address);
pmd_k = pmd_offset(pud_k, address);
if (!pmd_present(*pmd_k))
    goto no_context;
set_pmd(pmd, *pmd_k);
pte_k = pte_offset_kernel(pmd_k, address);
if (!pte_present(*pte_k))
    goto no_context;
return;
```

The pgd_paddr local variable is loaded with the physical address of the Page Global Directory of the current process, which is stored in the cr3 register.[*] The pgd local variable is then loaded with the linear address corresponding to pgd_paddr, and the pgd_k local variable is loaded with the linear address of the master kernel Page Global Directory.

---

[*] The kernel doesn't use current->mm->pgd to derive the address because this fault can occur anytime, even during a process switch.

If the master kernel Page Global Directory entry corresponding to the faulty linear address is null, the function jumps to the code at the `no_context` label (see the earlier section "Handling a Faulty Address Outside the Address Space"). Otherwise, the function looks at the master kernel Page Upper Directory entry and at the master kernel Page Middle Directory entry corresponding to the faulty linear address. Again, if either one of these entries is null, a jump is done to the `no_context` label. Otherwise, the master entry is copied into the corresponding entry of the process's Page Middle Directory.* Then the whole operation is repeated with the master Page Table entry.

Creating and Deleting a Process Address Space

Of the six typical cases mentioned earlier in the section "The Process's Address Space," in which a process gets new memory regions, the first one—issuing a `fork()` system call—requires the creation of a whole new address space for the child process. Conversely, when a process terminates, the kernel destroys its address space. In this section, we discuss how these two activities are performed by Linux.

Creating a Process Address Space

In the section "The clone(), fork(), and vfork() System Calls" in Chapter 3, we mentioned that the kernel invokes the `copy_mm()` function while creating a new process. This function creates the process address space by setting up all Page Tables and memory descriptors of the new process.

Each process usually has its own address space, but lightweight processes can be created by calling `clone()` with the `CLONE_VM` flag set. These processes share the same address space; that is, they are allowed to address the same set of pages.

Following the COW approach described earlier, traditional processes inherit the address space of their parent: pages stay shared as long as they are only read. When one of the processes attempts to write one of them, however, the page is duplicated; after some time, a forked process usually gets its own address space that is different from that of the parent process. Lightweight processes, on the other hand, use the address space of their parent process. Linux implements them simply by not duplicating address space. Lightweight processes can be created considerably faster than normal processes, and the sharing of pages can also be considered a benefit as long as the parent and children coordinate their accesses carefully.

---

* You might remember from the section "Paging in Linux" in Chapter 2 that if PAE is enabled then the Page Upper Directory entry cannot be null; otherwise, if PAE is disabled, setting the Page Middle Directory entry implicitly sets the Page Upper Directory entry, too.

If the new process has been created by means of the clone( ) system call and if the CLONE_VM flag of the flag parameter is set, copy_mm( ) gives the clone (tsk) the address space of its parent (current):

```
if (clone_flags & CLONE_VM) {
    atomic_inc(&current->mm->mm_users);
    spin_unlock_wait(&current->mm->page_table_lock);
    tsk->mm = current->mm;
    tsk->active_mm = current->mm;
    return 0;
}
```

Invoking the spin_unlock_wait( ) function ensures that, if the page table spin lock of the process is held by some other CPU, the page fault handler does not terminate until that lock is released. In fact, beside protecting the page tables, this spin lock must forbid the creation of new lightweight processes sharing the current->mm descriptor.

If the CLONE_VM flag is not set, copy_mm( ) must create a new address space (even though no memory is allocated within that address space until the process requests an address). The function allocates a new memory descriptor, stores its address in the mm field of the new process descriptor tsk, and copies the contents of current->mm into tsk->mm. It then changes a few fields of the new descriptor:

```
tsk->mm = kmem_cache_alloc(mm_cachep, SLAB_KERNEL);
memcpy(tsk->mm, current->mm, sizeof(*tsk->mm));
atomic_set(&tsk->mm->mm_users, 1);
atomic_set(&tsk->mm->mm_count, 1);
init_rwsem(&tsk->mm->mmap_sem);
tsk->mm->core_waiters = 0;
tsk->mm->page_table_lock = SPIN_LOCK_UNLOCKED;
tsk->mm->ioctx_list_lock = RW_LOCK_UNLOCKED;
tsk->mm->ioctx_list = NULL;
tsk->mm->default_kioctx = INIT_KIOCTX(tsk->mm->default_kioctx,
                                      *tsk->mm);
tsk->mm->free_area_cache = (TASK_SIZE/3+0xfff)&0xfffff000;
tsk->mm->pgd = pgd_alloc(tsk->mm);
tsk->mm->def_flags = 0;
```

Remember that the pgd_alloc( ) macro allocates a Page Global Directory for the new process.

The architecture-dependent init_new_context( ) function is then invoked: when dealing with 80×86 processors, this function checks whether the current process owns a customized Local Descriptor Table; if so, init_new_context( ) makes a copy of the Local Descriptor Table of current and adds it to the address space of tsk.

Finally, the dup_mmap( ) function is invoked to duplicate both the memory regions and the Page Tables of the parent process. This function inserts the new memory descriptor tsk->mm in the global list of memory descriptors. Then it scans the list of regions owned by the parent process, starting from the one pointed to by current->mm->mmap. It duplicates each vm_area_struct memory region descriptor encountered

and inserts the copy in the list of regions and in the red-black tree owned by the child process.

Right after inserting a new memory region descriptor, `dup_mmap( )` invokes `copy_page_range( )` to create, if necessary, the Page Tables needed to map the group of pages included in the memory region and to initialize the new Page Table entries. In particular, each page frame corresponding to a private, writable page (`VM_SHARED` flag off and `VM_MAYWRITE` flag on) is marked as read-only for both the parent and the child, so that it will be handled with the Copy On Write mechanism.

Deleting a Process Address Space

When a process terminates, the kernel invokes the `exit_mm( )` function to release the address space owned by that process:

```
mm_release(tsk, tsk->mm);
if (!(mm = tsk->mm)) /* kernel thread ? */
    return;
down_read(&mm->mmap_sem);
```

The `mm_release( )` function essentially wakes up all processes sleeping in the `tsk->vfork_done` completion (see the section "Completions" in Chapter 5). Typically, the corresponding wait queue is nonempty only if the exiting process was created by means of the `vfork( )` system call (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3).

If the process being terminated is not a kernel thread, the `exit_mm( )` function must release the memory descriptor and all related data structures. First of all, it checks whether the `mm->core_waiters` flag is set: if it does, then the process is dumping the contents of the memory to a core file. To avoid corruption in the core file, the function makes use of the `mm->core_done` and `mm->core_startup_done` completions to serialize the execution of the lightweight processes sharing the same memory descriptor `mm`.

Next, the function increases the memory descriptor's main usage counter, resets the `mm` field of the process descriptor, and puts the processor in lazy TLB mode (see "Handling the Hardware Cache and the TLB" in Chapter 2):

```
atomic_inc(&mm->mm_count);
spin_lock(tsk->alloc_lock);
tsk->mm = NULL;
up_read(&mm->map_sem);
enter_lazy_tlb(mm, current);
spin_unlock(tsk->alloc_lock);
mmput(mm);
```

Finally, the `mmput( )` function is invoked to release the Local Descriptor Table, the memory region descriptors, and the Page Tables. The memory descriptor itself, however, is not released, because `exit_mm( )` has increased the main usage counter. The

descriptor will be released by the `finish_task_switch()` function when the process being terminated will be effectively evicted from the local CPU (see the section "The schedule() Function" in Chapter 7).

Managing the Heap

Each Unix process owns a specific memory region called the *heap*, which is used to satisfy the process's dynamic memory requests. The `start_brk` and `brk` fields of the memory descriptor delimit the starting and ending addresses, respectively, of that region.

The following APIs can be used by the process to request and release dynamic memory:

`malloc(size)`
> Requests `size` bytes of dynamic memory; if the allocation succeeds, it returns the linear address of the first memory location.

`calloc(n,size)`
> Requests an array consisting of `n` elements of size `size`; if the allocation succeeds, it initializes the array components to 0 and returns the linear address of the first element.

`realloc(ptr,size)`
> Changes the size of a memory area previously allocated by `malloc()` or `calloc()`.

`free(addr)`
> Releases the memory region allocated by `malloc()` or `calloc()` that has an initial address of `addr`.

`brk(addr)`
> Modifies the size of the heap directly; the `addr` parameter specifies the new value of `current->mm->brk`, and the return value is the new ending address of the memory region (the process must check whether it coincides with the requested `addr` value).

`sbrk(incr)`
> Is similar to `brk()`, except that the `incr` parameter specifies the increment or decrement of the heap size in bytes.

The `brk()` function differs from the other functions listed because it is the only one implemented as a system call. All the other functions are implemented in the C library by using `brk()` and `mmap()`.*

When a process in User Mode invokes the `brk()` system call, the kernel executes the `sys_brk(addr)` function. This function first verifies whether the `addr` parameter falls

---

* The `realloc()` library function can also make use of the `mremap()` system call.

inside the memory region that contains the process code; if so, it returns immediately because the heap cannot overlap with memory region containing the process's code:

```
mm = current->mm;
down_write(&mm->mmap_sem);
if (addr < mm->end_code) {
out:
    up_write(&mm->mmap_sem);
    return mm->brk;
}
```

Because the brk( ) system call acts on a memory region, it allocates and deallocates whole pages. Therefore, the function aligns the value of addr to a multiple of PAGE_SIZE and compares the result with the value of the brk field of the memory descriptor:

```
newbrk = (addr + 0xfff) & 0xfffff000;
oldbrk = (mm->brk + 0xfff) & 0xfffff000;
if (oldbrk == newbrk) {
    mm->brk = addr;
    goto out;
}
```

If the process asked to shrink the heap, sys_brk( ) invokes the do_munmap( ) function to do the job and then returns:

```
if (addr <= mm->brk) {
    if (!do_munmap(mm, newbrk, oldbrk-newbrk))
        mm->brk = addr;
    goto out;
}
```

If the process asked to enlarge the heap, sys_brk( ) first checks whether the process is allowed to do so. If the process is trying to allocate memory outside its limit, the function simply returns the original value of mm->brk without allocating more memory:

```
rlim = current->signal->rlim[RLIMIT_DATA].rlim_cur;
if (rlim < RLIM_INFINITY && addr - mm->start_data > rlim)
    goto out;
```

The function then checks whether the enlarged heap would overlap some other memory region belonging to the process and, if so, returns without doing anything:

```
if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
    goto out;
```

If everything is OK, the do_brk( ) function is invoked. If it returns the oldbrk value, the allocation was successful and sys_brk( ) returns the value addr; otherwise, it returns the old mm->brk value:

```
if (do_brk(oldbrk, newbrk-oldbrk) == oldbrk)
    mm->brk = addr;
goto out;
```

The do_brk() function is actually a simplified version of do_mmap() that handles only anonymous memory regions. Its invocation might be considered equivalent to:

```
do_mmap(NULL, oldbrk, newbrk-oldbrk, PROT_READ|PROT_WRITE|PROT_EXEC,
        MAP_FIXED|MAP_PRIVATE, 0)
```

do_brk() is slightly faster than do_mmap(), because it avoids several checks on the memory region object fields by assuming that the memory region doesn't map a file on disk.

CHAPTER 10

System Calls

Operating systems offer processes running in User Mode a set of interfaces to inter-
act with hardware devices such as the CPU, disks, and printers. Putting an extra
layer between the application and the hardware has several advantages. First, it
makes programming easier by freeing users from studying low-level programming
characteristics of hardware devices. Second, it greatly increases system security,
because the kernel can check the accuracy of the request at the interface level before
attempting to satisfy it. Last but not least, these interfaces make programs more por-
table, because they can be compiled and executed correctly on every kernel that
offers the same set of interfaces.

Unix systems implement most interfaces between User Mode processes and hardware
devices by means of *system calls* issued to the kernel. This chapter examines in detail
how Linux implements system calls that User Mode programs issue to the kernel.

POSIX APIs and System Calls

Let's start by stressing the difference between an application programmer interface
(API) and a system call. The former is a function definition that specifies how to
obtain a given service, while the latter is an explicit request to the kernel made via a
software interrupt.

Unix systems include several libraries of functions that provide APIs to program-
mers. Some of the APIs defined by the *libc* standard C library refer to *wrapper rou-
tines* (routines whose only purpose is to issue a system call). Usually, each system call
has a corresponding wrapper routine, which defines the API that application pro-
grams should employ.

The converse is not true, by the way—an API does not necessarily correspond to a
specific system call. First of all, the API could offer its services directly in User Mode.
(For something abstract such as math functions, there may be no reason to make sys-
tem calls.) Second, a single API function could make several system calls. Moreover,

several API functions could make the same system call, but wrap extra functionality around it. For instance, in Linux, the `malloc( )`, `calloc( )`, and `free( )` APIs are implemented in the *libc* library. The code in this library keeps track of the allocation and deallocation requests and uses the `brk( )` system call to enlarge or shrink the process heap (see the section "Managing the Heap" in Chapter 9).

The POSIX standard refers to APIs and not to system calls. A system can be certified as POSIX-compliant if it offers the proper set of APIs to the application programs, no matter how the corresponding functions are implemented. As a matter of fact, several non-Unix systems have been certified as POSIX-compliant, because they offer all traditional Unix services in User Mode libraries.

From the programmer's point of view, the distinction between an API and a system call is irrelevant—the only things that matter are the function name, the parameter types, and the meaning of the return code. From the kernel designer's point of view, however, the distinction does matter because system calls belong to the kernel, while User Mode libraries don't.

Most wrapper routines return an integer value, whose meaning depends on the corresponding system call. A return value of −1 usually indicates that the kernel was unable to satisfy the process request. A failure in the system call handler may be caused by invalid parameters, a lack of available resources, hardware problems, and so on. The specific error code is contained in the `errno` variable, which is defined in the *libc* library.

Each error code is defined as a macro constant, which yields a corresponding positive integer value. The POSIX standard specifies the macro names of several error codes. In Linux, on 80×86 systems, these macros are defined in the header file *include/asm-i386/errno.h*. To allow portability of C programs among Unix systems, the *include/asm-i386/errno.h* header file is included, in turn, in the standard */usr/include/errno.h* C library header file. Other systems have their own specialized subdirectories of header files.

System Call Handler and Service Routines

When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function. As we will see in the next section, in the 80×86 architecture a Linux system call can be invoked in two different ways. The net result of both methods, however, is a jump to an assembly language function called the *system call handler*.

Because the kernel implements many different system calls, the User Mode process must pass a parameter called the *system call number* to identify the required system call; the `eax` register is used by Linux for this purpose. As we'll see in the section "Parameter Passing" later in this chapter, additional parameters are usually passed when invoking a system call.

All system calls return an integer value. The conventions for these return values are different from those for wrapper routines. In the kernel, positive or 0 values denote a successful termination of the system call, while negative values denote an error condition. In the latter case, the value is the negation of the error code that must be returned to the application program in the errno variable. The errno variable is not set or used by the kernel. Instead, the wrapper routines handle the task of setting this variable after a return from a system call.

The system call handler, which has a structure similar to that of the other exception handlers, performs the following operations:

- Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language).

- Handles the system call by invoking a corresponding C function called the *system call service routine*.

- Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language).

The name of the service routine associated with the *xyz( )* system call is usually sys_*xyz( )*; there are, however, a few exceptions to this rule.

Figure 10-1 illustrates the relationships between the application program that invokes a system call, the corresponding wrapper routine, the system call handler, and the system call service routine. The arrows denote the execution flow between the functions. The terms "SYSCALL" and "SYSEXIT" are placeholders for the actual assembly language instructions that switch the CPU, respectively, from User Mode to Kernel Mode and from Kernel Mode to User Mode.
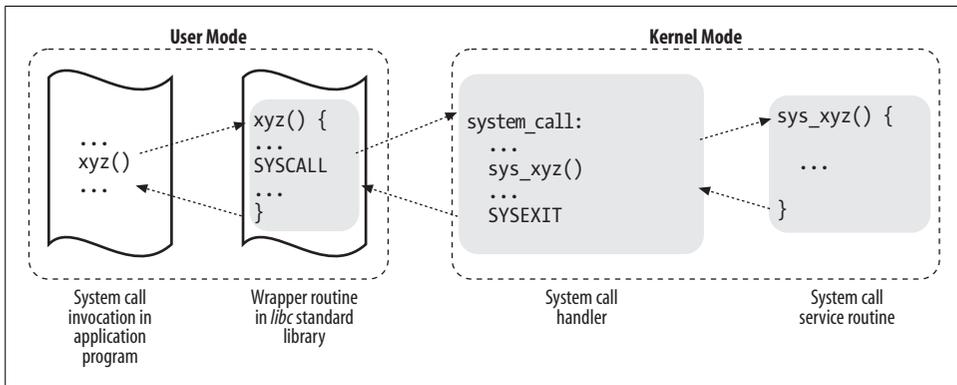
Figure 10-1. Invoking a system call

To associate each system call number with its corresponding service routine, the kernel uses a *system call dispatch table*, which is stored in the sys_call_table array and

has `NR_syscalls` entries (289 in the Linux 2.6.11 kernel). The nth entry contains the service routine address of the system call having number n.

The `NR_syscalls` macro is just a static limit on the maximum number of implementable system calls; it does not indicate the number of system calls actually implemented. Indeed, each entry of the dispatch table may contain the address of the `sys_ni_syscall( )` function, which is the service routine of the "nonimplemented" system calls; it just returns the error code `-ENOSYS`.

Entering and Exiting a System Call

Native applications[*] can invoke a system call in two different ways:

- By executing the `int $0x80` assembly language instruction; in older versions of the Linux kernel, this was the only way to switch from User Mode to Kernel Mode.
- By executing the `sysenter` assembly language instruction, introduced in the Intel Pentium II microprocessors; this instruction is now supported by the Linux 2.6 kernel.

Similarly, the kernel can exit from a system call—thus switching the CPU back to User Mode—in two ways:

- By executing the `iret` assembly language instruction.
- By executing the `sysexit` assembly language instruction, which was introduced in the Intel Pentium II microprocessors together with the `sysenter` instruction.

However, supporting two different ways to enter the kernel is not as simple as it might look, because:

- The kernel must support both older libraries that only use the `int $0x80` instruction and more recent ones that also use the `sysenter` instruction.
- A standard library that makes use of the `sysenter` instruction must be able to cope with older kernels that support only the `int $0x80` instruction.
- The kernel and the standard library must be able to run both on older processors that do not include the `sysenter` instruction and on more recent ones that include it.

We will see in the section "Issuing a System Call via the sysenter Instruction" later in this chapter how the Linux kernel solves these compatibility problems.

---

[*] As we will see in the section "Execution Domains" in Chapter 20, Linux can execute programs compiled for "foreign" operating systems. Therefore, the kernel offers a compatibility mode to enter a system call: User Mode processes executing iBCS and Solaris/x86 programs can enter the kernel by jumping into suitable call gates included in the default Local Descriptor Table (see the section "The Linux LDTs" in Chapter 2).

Issuing a System Call via the int $0x80 Instruction

The "traditional" way to invoke a system call makes use of the `int` assembly language instruction, which was discussed in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4.

The vector 128—in hexadecimal, `0x80`—is associated with the kernel entry point. The `trap_init()` function, invoked during kernel initialization, sets up the Interrupt Descriptor Table entry corresponding to vector 128 as follows:

```
set_system_gate(0x80, &system_call);
```

The call loads the following values into the gate descriptor fields (see the section "Interrupt, Trap, and System Gates" in Chapter 4):

Segment Selector
> The `__KERNEL_CS` Segment Selector of the kernel code segment.

Offset
> The pointer to the `system_call()` system call handler.

Type
> Set to 15. Indicates that the exception is a Trap and that the corresponding handler does not disable maskable interrupts.

DPL (Descriptor Privilege Level)
> Set to 3. This allows processes in User Mode to invoke the exception handler (see the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4).

Therefore, when a User Mode process issues an `int $0x80` instruction, the CPU switches into Kernel Mode and starts executing instructions from the `system_call` address.

The system_call() function

The `system_call()` function starts by saving the system call number and all the CPU registers that may be used by the exception handler on the stack—except for `eflags`, `cs`, `eip`, `ss`, and `esp`, which have already been saved automatically by the control unit (see the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4). The `SAVE_ALL` macro, which was already discussed in the section "I/O Interrupt Handling" in Chapter 4, also loads the Segment Selector of the kernel data segment in `ds` and `es`:

```
system_call:
  pushl %eax
  SAVE_ALL
  movl $0xffffe000, %ebx /* or 0xfffff000 for 4-KB stacks */
  andl %esp, %ebx
```

The function then stores the address of the `thread_info` data structure of the current process in `ebx` (see the section "Identifying a Process" in Chapter 3). This is done by

taking the value of the kernel stack pointer and rounding it up to a multiple of 4 or 8 KB (see the section "Identifying a Process" in Chapter 3).

Next, the `system_call( )` function checks whether either one of the `TIF_SYSCALL_TRACE` and `TIF_SYSCALL_AUDIT` flags included in the `flags` field of the `thread_info` structure is set—that is, whether the system call invocations of the executed program are being traced by a debugger. If this is the case, `system_call( )` invokes the `do_syscall_trace( )` function twice: once right before and once right after the execution of the system call service routine (as described later). This function stops `current` and thus allows the debugging process to collect information about it.

A validity check is then performed on the system call number passed by the User Mode process. If it is greater than or equal to the number of entries in the system call dispatch table, the system call handler terminates:

```
    cmpl $NR_syscalls, %eax
    jb nobadsys
    movl $(-ENOSYS), 24(%esp)
    jmp resume_userspace
nobadsys:
```

If the system call number is not valid, the function stores the `-ENOSYS` value in the stack location where the eax register has been saved—that is, at offset 24 from the current stack top. It then jumps to `resume_userspace` (see below). In this way, when the process resumes its execution in User Mode, it will find a negative return code in eax.

Finally, the specific service routine associated with the system call number contained in eax is invoked:

```
    call *sys_call_table(0, %eax, 4)
```

Because each entry in the dispatch table is 4 bytes long, the kernel finds the address of the service routine to be invoked by multiplying the system call number by 4, adding the initial address of the `sys_call_table` dispatch table, and extracting a pointer to the service routine from that slot in the table.

Exiting from the system call

When the system call service routine terminates, the `system_call( )` function gets its return code from eax and stores it in the stack location where the User Mode value of the eax register is saved:

```
    movl %eax, 24(%esp)
```

Thus, the User Mode process will find the return code of the system call in the eax register.

Then, the `system_call( )` function disables the local interrupts and checks the flags in the `thread_info` structure of `current`:

```
    cli
    movl 8(%ebp), %ecx
```

```
        testw $0xffff, %cx
        je restore_all
```

The `flags` field is at offset 8 in the `thread_info` structure; the mask `0xffff` selects the bits corresponding to all flags listed in Table 4-15 except `TIF_POLLING_NRFLAG`. If none of these flags is set, the function jumps to the `restore_all` label: as described in the section "Returning from Interrupts and Exceptions" in Chapter 4, this code restores the contents of the registers saved on the Kernel Mode stack and executes an `iret` assembly language instruction to resume the User Mode process. (You might refer to the flow diagram in Figure 4-6.)

If any of the flags is set, then there is some work to be done before returning to User Mode. If the `TIF_SYSCALL_TRACE` flag is set, the `system_call()` function invokes for the second time the `do_syscall_trace()` function, then jumps to the `resume_userspace` label. Otherwise, if the `TIF_SYSCALL_TRACE` flag is not set, the function jumps to the `work_pending` label.

As explained in the section "Returning from Interrupts and Exceptions" in Chapter 4, that code at the `resume_userspace` and `work_pending` labels checks for rescheduling requests, virtual-8086 mode, pending signals, and single stepping; then eventually a jump is done to the `restore_all` label to resume the execution of the User Mode process.

Issuing a System Call via the sysenter Instruction

The `int` assembly language instruction is inherently slow because it performs several consistency and security checks. (The instruction is described in detail in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4.)

The sysenter instruction, dubbed in Intel documentation as "Fast System Call," provides a faster way to switch from User Mode to Kernel Mode.

The sysenter instruction

The sysenter assembly language instruction makes use of three special registers that must be loaded with the following information:[*]

SYSENTER_CS_MSR
     The Segment Selector of the kernel code segment

SYSENTER_EIP_MSR
     The linear address of the kernel entry point

SYSENTER_ESP_MSR
     The kernel stack pointer

---

[*] "MSR" is an acronym for "Model-Specific Register" and denotes a register that is present only in some models of 80×86 microprocessors.

When the sysenter instruction is executed, the CPU control unit:

1. Copies the content of SYSENTER_CS_MSR into cs.

2. Copies the content of SYSENTER_EIP_MSR into eip.

3. Copies the content of SYSENTER_ESP_MSR into esp.

4. Adds 8 to the value of SYSENTER_CS_MSR, and loads this value into ss.

Therefore, the CPU switches to Kernel Mode and starts executing the first instruction of the kernel entry point. As we have seen in the section "The Linux GDT" in Chapter 2, the kernel stack segment coincides with the kernel data segment, and the corresponding descriptor follows the descriptor of the kernel code segment in the Global Descriptor Table; therefore, step 4 loads the proper Segment Selector in the ss register.

The three model-specific registers are initialized by the enable_sep_cpu() function, which is executed once by every CPU in the system during the initialization of the kernel. The function performs the following steps:

1. Writes the Segment Selector of the kernel code (__KERNEL_CS) in the SYSENTER_CS_MSR register.

2. Writes in the SYSENTER_CS_EIP register the linear address of the sysenter_entry() function described below.

3. Computes the linear address of the end of the local TSS, and writes this value in the SYSENTER_CS_ESP register.[*]

The setting of the SYSENTER_CS_ESP register deserves some comments. When a system call starts, the kernel stack is empty, thus the esp register should point to the end of the 4- or 8-KB memory area that includes the kernel stack and the descriptor of the current process (see Figure 3-2). The User Mode wrapper routine cannot properly set this register, because it does not know the address of this memory area; on the other hand, the value of the register must be set before switching to Kernel Mode. Therefore, the kernel initializes the register so as to encode the address of the Task State Segment of the local CPU. As we have described in step 3 of the __switch_to() function (see the section "Performing the Process Switch" in Chapter 3), at every process switch the kernel saves the kernel stack pointer of the current process in the esp0 field of the local TSS. Thus, the system call handler reads the esp register, computes the address of the esp0 field of the local TSS, and loads into the same esp register the proper kernel stack pointer.

---

[*] The encoding of the local TSS address written in SYSENTER_ESP_MSR is due to the fact that the register should point to a real stack, which grows towards lower address. In practice, initializing the register with any value would work, provided that it is possible to get the address of the local TSS from such a value.

The vsyscall page

A wrapper function in the *libc* standard library can make use of the sysenter instruction only if both the CPU and the Linux kernel support it.

This compatibility problem calls for a quite sophisticated solution. Essentially, in the initialization phase the sysenter_setup( ) function builds a page frame called *vsyscall page* containing a small ELF shared object (i.e., a tiny ELF dynamic library). When a process issues an execve( ) system call to start executing an ELF program, the code in the vsyscall page is dynamically linked to the process address space (see the section "The exec Functions" in Chapter 20). The code in the vsyscall page makes use of the best available instruction to issue a system call.

The sysenter_setup( ) function allocates a new page frame for the vsyscall page and associates its physical address with the FIX_VSYSCALL fix-mapped linear address (see the section "Fix-Mapped Linear Addresses" in Chapter 2). Then, the function copies in the page either one of two predefined ELF shared objects:

- If the CPU does not support sysenter, the function builds a vsyscall page that includes the code:

  ```
  __kernel_vsyscall:
      int $0x80
      ret
  ```

- Otherwise, if the CPU does support sysenter, the function builds a vsyscall page that includes the code:

  ```
  __kernel_vsyscall:
      pushl %ecx
      pushl %edx
      pushl %ebp
      movl %esp, %ebp
      sysenter
  ```

When a wrapper routine in the standard library must invoke a system call, it calls the __kernel_vsyscall( ) function, whatever it may be.

A final compatibility problem is due to old versions of the Linux kernel that do not support the sysenter instruction; in this case, of course, the kernel does not build the vsyscall page and the __kernel_vsyscall( ) function is not linked to the address space of the User Mode processes. When recent standard libraries recognize this fact, they simply execute the int $0x80 instruction to invoke the system calls.

Entering the system call

The sequence of steps performed when a system call is issued via the sysenter instruction is the following:

1. The wrapper routine in the standard library loads the system call number into the eax register and calls the __kernel_vsyscall( ) function.

2. The __kernel_vsyscall() function saves on the User Mode stack the contents of ebp, edx, and ecx (these registers are going to be used by the system call handler), copies the user stack pointer in ebp, then executes the sysenter instruction.

3. The CPU switches from User Mode to Kernel Mode, and the kernel starts executing the sysenter_entry() function (pointed to by the SYSENTER_EIP_MSR register).

4. The sysenter_entry() assembly language function performs the following steps:

   a. Sets up the kernel stack pointer:

      ```
      movl -508(%esp), %esp
      ```

      Initially, the esp register points to the first location after the local TSS, which is 512bytes long. Therefore, the instruction loads in the esp register the contents of the field at offset 4 in the local TSS, that is, the contents of the esp0 field. As already explained, the esp0 field always stores the kernel stack pointer of the current process.

   b. Enables local interrupts:

      ```
      sti
      ```

   c. Saves in the Kernel Mode stack the Segment Selector of the user data segment, the current user stack pointer, the eflags register, the Segment Selector of the user code segment, and the address of the instruction to be executed when exiting from the system call:

      ```
      pushl $(__USER_DS)
      pushl %ebp
      pushfl
      pushl $(__USER_CS)
      pushl $SYSENTER_RETURN
      ```

      Observe that these instructions emulate some operations performed by the int assembly language instruction (steps 5c and 7 in the description of int in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4).

   d. Restores in ebp the original value of the register passed by the wrapper routine:

      ```
      movl (%ebp), %ebp
      ```

      This instruction does the job, because __kernel_vsyscall() saved on the User Mode stack the original value of ebp and then loaded in ebp the current value of the user stack pointer.

   e. Invokes the system call handler by executing a sequence of instructions identical to that starting at the system_call label described in the earlier section "Issuing a System Call via the int $0x80 Instruction."

Exiting from the system call

When the system call service routine terminates, the sysenter_entry() function executes essentially the same operations as the system_call() function (see previous

section). First, it gets the return code of the system call service routine from `eax` and stores it in the kernel stack location where the User Mode value of the eax register is saved. Then, the function disables the local interrupts and checks the flags in the `thread_info` structure of `current`.

If any of the flags is set, then there is some work to be done before returning to User Mode. In order to avoid code duplication, this case is handled exactly as in the `system_call()` function, thus the function jumps to the `resume_userspace` or `work_pending` labels (see flow diagram in Figure 4-6 in Chapter 4). Eventually, the `iret` assembly language instruction fetches from the Kernel Mode stack the five arguments saved in step 4c by the `sysenter_entry()` function, and thus switches the CPU back to User Mode and starts executing the code at the `SYSENTER_RETURN` label (see below).

If the `sysenter_entry()` function determines that the flags are cleared, it performs a quick return to User Mode:

```
movl 40(%esp), %edx
movl 52(%esp), %ecx
xorl %ebp, %ebp
sti
sysexit
```

The `edx` and `ecx` registers are loaded with a couple of the stack values saved by `sysenter_entry()` in step 4c in the previos section: `edx` gets the address of the `SYSENTER_RETURN` label, while `ecx` gets the current user data stack pointer.

The sysexit instruction

The `sysexit` assembly language instruction is the companion of `sysenter`: it allows a fast switch from Kernel Mode to User Mode. When the instruction is executed, the CPU control unit performs the following steps:

1. Adds 16 to the value in the `SYSENTER_CS_MSR` register, and loads the result in the `cs` register.
2. Copies the content of the `edx` register into the `eip` register.
3. Adds 24 to the value in the `SYSENTER_CS_MSR` register, and loads the result in the `ss` register.
4. Copies the content of the `ecx` register into the `esp` register.

Because the `SYSENTER_CS_MSR` register is loaded with the Segment Selector of the kernel code, the `cs` register is loaded with the Segment Selector of the user code, while the `ss` register is loaded with the Segment Selector of the user data segment (see the section "The Linux GDT" in Chapter 2).

As a result, the CPU switches from Kernel Mode to User Mode and starts executing the instruction whose address is stored in the `edx` register.

The SYSENTER_RETURN code

The code at the SYSENTER_RETURN label is stored in the vsyscall page, and it is executed when a system call entered via sysenter is being terminated, either by the iret instruction or the sysexit instruction.

The code simply restores the original contents of the ebp, edx, and ecx registers saved in the User Mode stack, and returns the control to the wrapper routine in the standard library:

```
SYSENTER_RETURN:
    popl %ebp
    popl %edx
    popl %ecx
    ret
```

Parameter Passing

Like ordinary functions, system calls often require some input/output parameters, which may consist of actual values (i.e., numbers), addresses of variables in the address space of the User Mode process, or even addresses of data structures including pointers to User Mode functions (see the section "System Calls Related to Signal Handling" in Chapter 11).

Because the system_call() and the sysenter_entry() functions are the common entry points for all system calls in Linux, each of them has at least one parameter: the system call number passed in the eax register. For instance, if an application program invokes the fork() wrapper routine, the eax register is set to 2 (i.e., __NR_fork) before executing the int $0x80 or sysenter assembly language instruction. Because the register is set by the wrapper routines included in the *libc* library, programmers do not usually care about the system call number.

The fork() system call does not require other parameters. However, many system calls do require additional parameters, which must be explicitly passed by the application program. For instance, the mmap() system call may require up to six additional parameters (besides the system call number).

The parameters of ordinary C functions are usually passed by writing their values in the active program stack (either the User Mode stack or the Kernel Mode stack). Because system calls are a special kind of function that cross over from user to kernel land, neither the User Mode or the Kernel Mode stacks can be used. Rather, system call parameters are written in the CPU registers before issuing the system call. The kernel then copies the parameters stored in the CPU registers onto the Kernel Mode stack before invoking the system call service routine, because the latter is an ordinary C function.

Why doesn't the kernel copy parameters directly from the User Mode stack to the Kernel Mode stack? First of all, working with two stacks at the same time is complex;

second, the use of registers makes the structure of the system call handler similar to that of other exception handlers.

However, to pass parameters in registers, two conditions must be satisfied:

- The length of each parameter cannot exceed the length of a register (32 bits).*
- The number of parameters must not exceed six, besides the system call number passed in eax, because 80×86 processors have a very limited number of registers.

The first condition is always true because, according to the POSIX standard, large parameters that cannot be stored in a 32-bit register must be passed by reference. A typical example is the settimeofday( ) system call, which must read a 64-bit structure.

However, system calls that require more than six parameters exist. In such cases, a single register is used to point to a memory area in the process address space that contains the parameter values. Of course, programmers do not have to care about this workaround. As with every C function call, parameters are automatically saved on the stack when the wrapper routine is invoked. This routine will find the appropriate way to pass the parameters to the kernel.

The registers used to store the system call number and its parameters are, in increasing order, eax (for the system call number), ebx, ecx, edx, esi, edi, and ebp. As seen before, system_call( ) and sysenter_entry( ) save the values of these registers on the Kernel Mode stack by using the SAVE_ALL macro. Therefore, when the system call service routine goes to the stack, it finds the return address to system_call( ) or to sysenter_entry( ), followed by the parameter stored in ebx (the first parameter of the system call), the parameter stored in ecx, and so on (see the section "Saving the registers for the interrupt handler" in Chapter 4). This stack configuration is exactly the same as in an ordinary function call, and therefore the service routine can easily refer to its parameters by using the usual C-language constructs.

Let's look at an example. The sys_write( ) service routine, which handles the write( ) system call, is declared as:

```
int sys_write (unsigned int fd, const char * buf, unsigned int count)
```

The C compiler produces an assembly language function that expects to find the fd, buf, and count parameters on top of the stack, right below the return address, in the locations used to save the contents of the ebx, ecx, and edx registers, respectively.

In a few cases, even if the system call doesn't use any parameters, the corresponding service routine needs to know the contents of the CPU registers right before the system call was issued. For example, the do_fork( ) function that implements fork( ) needs to know the value of the registers in order to duplicate them in the child process thread field (see the section "The thread field" in Chapter 3). In these cases, a

---

* We refer, as usual, to the 32-bit architecture of the 80×86 processors. The discussion in this section does not apply to 64-bit architectures.

single parameter of type `pt_regs` allows the service routine to access the values saved in the Kernel Mode stack by the `SAVE_ALL` macro (see the section "The do_IRQ() function" in Chapter 4):

```
int sys_fork (struct pt_regs regs)
```

The return value of a service routine must be written into the `eax` register. This is automatically done by the C compiler when a `return n;` instruction is executed.

Verifying the Parameters

All system call parameters must be carefully checked before the kernel attempts to satisfy a user request. The type of check depends both on the system call and on the specific parameter. Let's go back to the `write()` system call introduced before: the `fd` parameter should be a file descriptor that identifies a specific file, so `sys_write()` must check whether `fd` really is a file descriptor of a file previously opened and whether the process is allowed to write into it (see the section "File-Handling System Calls" in Chapter 1). If any of these conditions are not true, the handler must return a negative value—in this case, the error code –`EBADF`.

One type of checking, however, is common to all system calls. Whenever a parameter specifies an address, the kernel must check whether it is inside the process address space. There are two possible ways to perform this check:

- Verify that the linear address belongs to the process address space and, if so, that the memory region including it has the proper access rights.
- Verify just that the linear address is lower than `PAGE_OFFSET` (i.e., that it doesn't fall within the range of interval addresses reserved to the kernel).

Early Linux kernels performed the first type of checking. But it is quite time consuming because it must be executed for each address parameter included in a system call; furthermore, it is usually pointless because faulty programs are not very common.

Therefore, starting with Version 2.2, Linux employs the second type of checking. This is much more efficient because it does not require any scan of the process memory region descriptors. Obviously, this is a very coarse check: verifying that the linear address is smaller than `PAGE_OFFSET` is a necessary but not sufficient condition for its validity. But there's no risk in confining the kernel to this limited kind of check because other errors will be caught later.

The approach followed is thus to defer the real checking until the last possible moment—that is, until the Paging Unit translates the linear address into a physical one. We will discuss in the section "Dynamic Address Checking: The Fix-up Code," later in this chapter, how the Page Fault exception handler succeeds in detecting those bad addresses issued in Kernel Mode that were passed as parameters by User Mode processes.

One might wonder at this point why the coarse check is performed at all. This type of checking is actually crucial to preserve both process address spaces and the kernel address space from illegal accesses. We saw in Chapter 2 that the RAM is mapped starting from PAGE_OFFSET. This means that kernel routines are able to address all pages present in memory. Thus, if the coarse check were not performed, a User Mode process might pass an address belonging to the kernel address space as a parameter and then be able to read or write every page present in memory without causing a Page Fault exception.

The check on addresses passed to system calls is performed by the access_ok() macro, which acts on two parameters: addr and size. The macro checks the address interval delimited by addr and addr + size - 1. It is essentially equivalent to the following C function:

```
int access_ok(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a ||
        a + size > current_thread_info()->addr_limit.seg)
        return 0;
    return 1;
}
```

The function first verifies whether addr + size, the highest address to be checked, is larger than $2^{32}-1$; because unsigned long integers and pointers are represented by the GNU C compiler (gcc) as 32-bit numbers, this is equivalent to checking for an overflow condition. The function also checks whether addr + size exceeds the value stored in the addr_limit.seg field of the thread_info structure of current. This field usually has the value PAGE_OFFSET for normal processes and the value 0xffffffff for kernel threads. The value of the addr_limit.seg field can be dynamically changed by the get_fs and set_fs macros; this allows the kernel to bypass the security checks made by access_ok(), so that it can invoke system call service routines, directly passing to them addresses in the kernel data segment.

The verify_area() function performs the same check as the access_ok() macro; although this function is considered obsolete, it is still widely used in the source code.

Accessing the Process Address Space

System call service routines often need to read or write data contained in the process's address space. Linux includes a set of macros that make this access easier. We'll describe two of them, called get_user() and put_user(). The first can be used to read 1, 2, or 4 consecutive bytes from an address, while the second can be used to write data of those sizes into an address.

Each function accepts two arguments, a value x to transfer and a variable ptr. The second variable also determines how many bytes to transfer. Thus, in get_user(x,ptr),

the size of the variable pointed to by ptr causes the function to expand into a __get_user_1( ), __get_user_2( ), or __get_user_4( ) assembly language function. Let's consider one of them, __get_user_2( ):

```
__get_user_2:
    addl $1, %eax
    jc bad_get_user
    movl $0xffffe000, %edx /* or 0xfffff000 for 4-KB stacks */
    andl %esp, %edx
    cmpl 24(%edx), %eax
    jae bad_get_user
2:  movzwl -1(%eax), %edx
    xorl %eax, %eax
    ret
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
```

The eax register contains the address ptr of the first byte to be read. The first six instructions essentially perform the same checks as the access_ok( ) macro: they ensure that the 2 bytes to be read have addresses less than 4 GB as well as less than the addr_limit.seg field of the current process. (This field is stored at offset 24 in the thread_info structure of current, which appears in the first operand of the cmpl instruction.)

If the addresses are valid, the function executes the movzwl instruction to store the data to be read in the two least significant bytes of edx register while setting the high-order bytes of edx to 0; then it sets a 0 return code in eax and terminates. If the addresses are not valid, the function clears edx, sets the -EFAULT value into eax, and terminates.

The put_user(x,ptr) macro is similar to the one discussed before, except it writes the value x into the process address space starting from address ptr. Depending on the size of x, it invokes either the __put_user_asm( ) macro (size of 1, 2, or 4 bytes) or the __put_user_u64( ) macro (size of 8 bytes). Both macros return the value 0 in the eax register if they succeed in writing the value, and -EFAULT otherwise.

Several other functions and macros are available to access the process address space in Kernel Mode; they are listed in Table 10-1. Notice that many of them also have a variant prefixed by two underscores (__). The ones without initial underscores take extra time to check the validity of the linear address interval requested, while the ones with the underscores bypass that check. Whenever the kernel must repeatedly access the same memory area in the process address space, it is more efficient to check the address once at the start and then access the process area without making any further checks.

Table 10-1. Functions and macros that access the process address space

Function	Action
get_user __get_user	Reads an integer value from user space (1, 2, or 4 bytes)
put_user __put_user	Writes an integer value to user space (1, 2, or 4 bytes)
copy_from_user __copy_from_user	Copies a block of arbitrary size from user space
copy_to_user __copy_to_user	Copies a block of arbitrary size to user space
strncpy_from_user __strncpy_from_user	Copies a null-terminated string from user space
strlen_user strnlen_user	Returns the length of a null-terminated string in user space
clear_user __clear_user	Fills a memory area in user space with zeros

Dynamic Address Checking: The Fix-up Code

As seen previously, access_ok( ) makes a coarse check on the validity of linear addresses passed as parameters of a system call. This check only ensures that the User Mode process is not attempting to fiddle with the kernel address space; however, the linear addresses passed as parameters still might not belong to the process address space. In this case, a Page Fault exception will occur when the kernel tries to use any of such bad addresses.

Before describing how the kernel detects this type of error, let's specify the three cases in which Page Fault exceptions may occur in Kernel Mode. These cases must be distinguished by the Page Fault handler, because the actions to be taken are quite different.

1. The kernel attempts to address a page belonging to the process address space, but either the corresponding page frame does not exist or the kernel tries to write a read-only page. In these cases, the handler must allocate and initialize a new page frame (see the sections "Demand Paging" and "Copy On Write" in Chapter 9).

2. The kernel addresses a page belonging to its address space, but the corresponding Page Table entry has not yet been initialized (see the section "Handling Noncontiguous Memory Area Accesses" in Chapter 9). In this case, the kernel must properly set up some entries in the Page Tables of the current process.

3. Some kernel functions include a programming bug that causes the exception to be raised when that program is executed; alternatively, the exception might be caused by a transient hardware error. When this occurs, the handler must perform a kernel oops (see the section "Handling a Faulty Address Inside the Address Space" in Chapter 9).

4. The case introduced in this chapter: a system call service routine attempts to read or write into a memory area whose address has been passed as a system call parameter, but that address does not belong to the process address space.

The Page Fault handler can easily recognize the first case by determining whether the faulty linear address is included in one of the memory regions owned by the process.

It is also able to detect the second case by checking whether the corresponding master kernel Page Table entry includes a proper non-null entry that maps the address. Let's now explain how the handler distinguishes the remaining two cases.

The Exception Tables

The key to determining the source of a Page Fault lies in the narrow range of calls that the kernel uses to access the process address space. Only the small group of functions and macros described in the previous section are used to access this address space; thus, if the exception is caused by an invalid parameter, the instruction that caused it *must* be included in one of the functions or else be generated by expanding one of the macros. The number of the instructions that address user space is fairly small.

Therefore, it does not take much effort to put the address of each kernel instruction that accesses the process address space into a structure called the *exception table*. If we succeed in doing this, the rest is easy. When a Page Fault exception occurs in Kernel Mode, the do_page_fault( ) handler examines the exception table: if it includes the address of the instruction that triggered the exception, the error is caused by a bad system call parameter; otherwise, it is caused by a more serious bug.

Linux defines several exception tables. The main exception table is automatically generated by the C compiler when building the kernel program image. It is stored in the __ex_table section of the kernel code segment, and its starting and ending addresses are identified by two symbols produced by the C compiler: __start___ex_table and __stop___ex_table.

Moreover, each dynamically loaded module of the kernel (see Appendix B) includes its own local exception table. This table is automatically generated by the C compiler when building the module image, and it is loaded into memory when the module is inserted in the running kernel.

Each entry of an exception table is an exception_table_entry structure that has two fields:

insn
> The linear address of an instruction that accesses the process address space

fixup
> The address of the assembly language code to be invoked when a Page Fault exception triggered by the instruction located at insn occurs

The fixup code consists of a few assembly language instructions that solve the problem triggered by the exception. As we will see later in this section, the fix usually consists of inserting a sequence of instructions that forces the service routine to return an error code to the User Mode process. These instructions, which are usually defined in the same macro or function that accesses the process address space, are placed by the C compiler into a separate section of the kernel code segment called .fixup.

The search_exception_tables( ) function is used to search for a specified address in all exception tables: if the address is included in a table, the function returns a pointer to the corresponding exception_table_entry structure; otherwise, it returns NULL. Thus the Page Fault handler do_page_fault( ) executes the following statements:

```
if ((fixup = search_exception_tables(regs->eip))) {
    regs->eip = fixup->fixup;
    return 1;
}
```

The regs->eip field contains the value of the eip register saved on the Kernel Mode stack when the exception occurred. If the value in the register (the instruction pointer) is in an exception table, do_page_fault( ) replaces the saved value with the address found in the entry returned by search_exception_tables( ). Then the Page Fault handler terminates and the interrupted program resumes with execution of the fixup code.

Generating the Exception Tables and the Fixup Code

The GNU Assembler .section directive allows programmers to specify which section of the executable file contains the code that follows. As we will see in Chapter 20, an executable file includes a code segment, which in turn may be subdivided into sections. Thus, the following assembly language instructions add an entry into an exception table; the "a" attribute specifies that the section must be loaded into memory together with the rest of the kernel image:

```
.section __ex_table, "a"
    .long faulty_instruction_address, fixup_code_address
.previous
```

The .previous directive forces the assembler to insert the code that follows into the section that was active when the last .section directive was encountered.

Let's consider again the __get_user_1( ), __get_user_2( ), and __get_user_4( ) functions mentioned before. The instructions that access the process address space are those labeled as 1, 2, and 3:

```
__get_user_1:
    [...]
1:  movzbl (%eax), %edx
    [...]
__get_user_2:
    [...]
2:  movzwl -1(%eax), %edx
    [...]
__get_user_4:
    [...]
3:  movl -3(%eax), %edx
    [...]
bad_get_user:
    xorl %edx, %edx
```

```
    movl $-EFAULT, %eax
    ret
.section __ex_table,"a"
    .long 1b, bad_get_user
    .long 2b, bad_get_user
    .long 3b, bad_get_user
.previous
```

Each exception table entry consists of two labels. The first one is a numeric label with a b suffix to indicate that the label is "backward;" in other words, it appears in a previous line of the program. The fixup code is common to the three functions and is labeled as bad_get_user. If a Page Fault exception is generated by the instructions at label 1, 2, or 3, the fixup code is executed. It simply returns an –EFAULT error code to the process that issued the system call.

Other kernel functions that act in the User Mode address space use the fixup code technique. Consider, for instance, the strlen_user(string) macro. This macro returns either the length of a null-terminated string passed as a parameter in a system call or the value 0 on error. The macro essentially yields the following assembly language instructions:

```
    movl $0, %eax
    movl $0x7ffffffff, %ecx
    movl %ecx, %ebx
    movl string, %edi
0:  repne; scasb
    subl %ecx, %ebx
    movl %ebx, %eax
1:
.section .fixup,"ax"
2:  xorl %eax, %eax
    jmp 1b
.previous
.section __ex_table,"a"
    .long 0b, 2b
.previous
```

The ecx and ebx registers are initialized with the 0x7ffffff value, which represents the maximum allowed length for the string in the User Mode address space. The repne;scasb assembly language instructions iteratively scan the string pointed to by the edi register, looking for the value 0 (the end of string \0 character) in eax. Because scasb decreases the ecx register at each iteration, the eax register ultimately stores the total number of bytes scanned in the string (that is, the length of the string).

The fixup code of the macro is inserted into the .fixup section. The "ax" attributes specify that the section must be loaded into memory and that it contains executable code. If a Page Fault exception is generated by the instructions at label 0, the fixup code is executed; it simply loads the value 0 in eax—thus forcing the macro to return a 0 error code instead of the string length—and then jumps to the 1 label, which corresponds to the instruction following the macro.

The second .section directive adds an entry containing the address of the repne; scasb instruction and the address of the corresponding fixup code in the __ex_table section.

Kernel Wrapper Routines

Although system calls are used mainly by User Mode processes, they can also be invoked by kernel threads, which cannot use library functions. To simplify the declarations of the corresponding wrapper routines, Linux defines a set of seven macros called _syscall0 through _syscall6.

In the name of each macro, the numbers 0 through 6 correspond to the number of parameters used by the system call (excluding the system call number). The macros are used to declare wrapper routines that are not already included in the *libc* standard library (for instance, because the Linux system call is not yet supported by the library); however, they cannot be used to define wrapper routines for system calls that have more than six parameters (excluding the system call number) or for system calls that yield nonstandard return values.

Each macro requires exactly $2 + 2 \times n$ parameters, with n being the number of parameters of the system call. The first two parameters specify the return type and the name of the system call; each additional pair of parameters specifies the type and the name of the corresponding system call parameter. Thus, for instance, the wrapper routine of the fork( ) system call may be generated by:

```
_syscall0(int,fork)
```

while the wrapper routine of the write( ) system call may be generated by:

```
_syscall3(int,write,int,fd,const char *,buf,unsigned int,count)
```

In the latter case, the macro yields the following code:

```
int write(int fd,const char * buf,unsigned int count)
{
    long __res;
    asm("int $0x80"
        : "=a" (__res)
        : "0" (__NR_write), "b" ((long)fd),
          "c" ((long)buf), "d" ((long)count));
    if ((unsigned long)__res >= (unsigned long)-129) {
        errno = -__res;
        __res = -1;
    }
    return (int) __res;
}
```

The __NR_write macro is derived from the second parameter of _syscall3; it expands into the system call number of write( ). When compiling the preceding function, the following assembly language code is produced:

```
write:
     pushl %ebx               ; push ebx into stack
     movl 8(%esp), %ebx       ; put first parameter in ebx
     movl 12(%esp), %ecx      ; put second parameter in ecx
     movl 16(%esp), %edx      ; put third parameter in edx
     movl $4, %eax            ; put __NR_write in eax
     int $0x80                ; invoke system call
     cmpl $-125, %eax         ; check return code
     jbe .L1                  ; if no error, jump
     negl %eax                ; complement the value of eax
     movl %eax, errno         ; put result in errno
     movl $-1, %eax           ; set eax to -1
.L1: popl %ebx               ; pop ebx from stack
     ret                      ; return to calling program
```

Notice how the parameters of the write( ) function are loaded into the CPU registers before the int $0x80 instruction is executed. The value returned in eax must be interpreted as an error code if it lies between −1 and −129 (the kernel assumes that the largest error code defined in *include/generic/errno.h* is 129). If this is the case, the wrapper routine stores the value of -eax in errno and returns the value −1; otherwise, it returns the value of eax.

CHAPTER 11

Signals

Signals were introduced by the first Unix systems to allow interactions between User Mode processes; the kernel also uses them to notify processes of system events. Signals have been around for 30 years with only minor changes.

The first sections of this chapter examine in detail how signals are handled by the Linux kernel, then we discuss the system calls that allow processes to exchange signals.

The Role of Signals

A *signal* is a very short message that may be sent to a process or a group of processes. The only information given to the process is usually a number identifying the signal; there is no room in standard signals for arguments, a message, or other accompanying information.

A set of macros whose names start with the prefix SIG is used to identify signals; we have already made a few references to them in previous chapters. For instance, the SIGCHLD macro was mentioned in the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3. This macro, which expands into the value 17 in Linux, yields the identifier of the signal that is sent to a parent process when a child stops or terminates. The SIGSEGV macro, which expands into the value 11, was mentioned in the section "Page Fault Exception Handler" in Chapter 9; it yields the identifier of the signal that is sent to a process when it makes an invalid memory reference.

Signals serve two main purposes:

* To make a process aware that a specific event has occurred
* To cause a process to execute a *signal handler* function included in its code

Of course, the two purposes are not mutually exclusive, because often a process must react to some event by executing a specific routine.

Table 11-1 lists the first 31 signals handled by Linux 2.6 for the 80×86 architecture (some signal numbers, such those associated with SIGCHLD or SIGSTOP, are

architecture-dependent; furthermore, some signals such as SIGSTKFLT are defined only for specific architectures). The meanings of the default actions are described in the next section.

Table 11-1. The first 31 signals in Linux/i386

#	Signal name	Default action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard	Yes
3	SIGQUIT	Dump	Quit from keyboard	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timerclock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No
17	SIGCHLD	Ignore	Child process stopped or terminated, or got signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No

Table 11-1. The first 31 signals in Linux/i386 (continued)

#	Signal name	Default action	Comment	POSIX
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No

Besides the *regular signals* described in this table, the POSIX standard has introduced a new class of signals denoted as *real-time signals*; their signal numbers range from 32 to 64 on Linux. They mainly differ from regular signals because they are always queued so that multiple signals sent will be received. On the other hand, regular signals of the same kind are not queued: if a regular signal is sent many times in a row, just one of them is delivered to the receiving process. Although the Linux kernel does not use real-time signals, it fully supports the POSIX standard by means of several specific system calls.

A number of system calls allow programmers to send signals and determine how their processes respond to the signals they receive. Table 11-2 summarizes these calls; their behavior is described in detail in the later section "System Calls Related to Signal Handling."

Table 11-2. The most significant system calls related to signals

System call	Description
kill( )	Send a signal to a thread group
tkill( )	Send a signal to a process
tgkill( )	Send a signal to a process in a specific thread group
sigaction( )	Change the action associated with a signal
signal( )	Similar to sigaction( )
sigpending( )	Check whether there are pending signals
sigprocmask( )	Modify the set of blocked signals
sigsuspend( )	Wait for a signal
rt_sigaction( )	Change the action associated with a real-time signal
rt_sigpending( )	Check whether there are pending real-time signals
rt_sigprocmask( )	Modify the set of blocked real-time signals
rt_sigqueueinfo( )	Send a real-time signal to a thread group
rt_sigsuspend( )	Wait for a real-time signal
rt_sigtimedwait( )	Similar to rt_sigsuspend( )

An important characteristic of signals is that they may be sent at any time to a process whose state is usually unpredictable. Signals sent to a process that is not currently executing must be saved by the kernel until that process resumes execution. Blocking a signal (described later) requires that delivery of the signal be held off until it is later unblocked, which exacerbates the problem of signals being raised before they can be delivered.

Therefore, the kernel distinguishes two different phases related to signal transmission:

Signal generation
> The kernel updates a data structure of the destination process to represent that a new signal has been sent.

Signal delivery
> The kernel forces the destination process to react to the signal by changing its execution state, by starting the execution of a specified signal handler, or both.

Each signal generated can be delivered once, at most. Signals are consumable resources: once they have been delivered, all process descriptor information that refers to their previous existence is canceled.

Signals that have been generated but not yet delivered are called *pending signals*. At any time, only one pending signal of a given type may exist for a process; additional pending signals of the same type to the same process are not queued but simply discarded. Real-time signals are different, though: there can be several pending signals of the same type.

In general, a signal may remain pending for an unpredictable amount of time. The following factors must be taken into consideration:

- Signals are usually delivered only to the currently running process (that is, to the `current` process).

- Signals of a given type may be selectively *blocked* by a process (see the later section "Modifying the Set of Blocked Signals"). In this case, the process does not receive the signal until it removes the block.

- When a process executes a signal-handler function, it usually *masks* the corresponding signal—i.e., it automatically blocks the signal until the handler terminates. A signal handler therefore cannot be interrupted by another occurrence of the handled signal, and the function doesn't need to be reentrant.

Although the notion of signals is intuitive, the kernel implementation is rather complex. The kernel must:

- Remember which signals are blocked by each process.

- When switching from Kernel Mode to User Mode, check whether a signal for a process has arrived. This happens at almost every timer interrupt (roughly every millisecond).

- Determine whether the signal can be ignored. This happens when all of the following conditions are fulfilled:

  — The destination process is not traced by another process (the `PT_PTRACED` flag in the process descriptor `ptrace` field is equal to 0).[*]

---

[*] If a process receives a signal while it is being traced, the kernel stops the process and notifies the tracing process by sending a `SIGCHLD` signal to it. The tracing process may, in turn, resume execution of the traced process by means of a `SIGCONT` signal.

— The signal is not blocked by the destination process.

— The signal is being ignored by the destination process (either because the process explicitly ignored it or because the process did not change the default action of the signal and that action is "ignore").

• Handle the signal, which may require switching the process to a handler function at any point during its execution and restoring the original execution context after the function returns.

Moreover, Linux must take into account the different semantics for signals adopted by BSD and System V; furthermore, it must comply with the rather cumbersome POSIX requirements.

Actions Performed upon Delivering a Signal

There are three ways in which a process can respond to a signal:

1. Explicitly ignore the signal.

2. Execute the *default action* associated with the signal (see Table 11-1). This action, which is predefined by the kernel, depends on the signal type and may be any one of the following:

   Terminate
   > The process is terminated (killed).

   Dump
   > The process is terminated (killed) and a `core` file containing its execution context is created, if possible; this file may be used for debug purposes.

   Ignore
   > The signal is ignored.

   Stop
   > The process is stopped—i.e., put in the `TASK_STOPPED` state (see the section "Process State" in Chapter 3).

   Continue
   > If the process was stopped (`TASK_STOPPED`), it is put into the `TASK_RUNNING` state.

3. Catch the signal by invoking a corresponding signal-handler function.

Notice that blocking a signal is different from ignoring it. A signal is not delivered as long as it is blocked; it is delivered only after it has been unblocked. An ignored signal is always delivered, and there is no further action.

The `SIGKILL` and `SIGSTOP` signals cannot be ignored, caught, or blocked, and their default actions must always be executed. Therefore, `SIGKILL` and `SIGSTOP` allow a user

with appropriate privileges to terminate and to stop, respectively, every process,[*] regardless of the defenses taken by the program it is executing.

A signal is *fatal* for a given process if delivering the signal causes the kernel to kill the process. The SIGKILL signal is always fatal; moreover, each signal whose default action is "Terminate" and which is not caught by a process is also fatal for that process. Notice, however, that a signal caught by a process and whose corresponding signal-handler function terminates the process is not fatal, because the process chose to terminate itself rather than being killed by the kernel.

POSIX Signals and Multithreaded Applications

The POSIX 1003.1 standard has some stringent requirements for signal handling of multithreaded applications:

- Signal handlers must be shared among all threads of a multithreaded application; however, each thread must have its own mask of pending and blocked signals.

- The kill() and sigqueue() POSIX library functions (see the later section "System Calls Related to Signal Handling") must send signals to whole multithreaded applications, not to a specific thread. The same holds for all signals (such as SIGCHLD, SIGINT, or SIGQUIT) generated by the kernel.

- Each signal sent to a multithreaded application will be delivered to just one thread, which is arbitrarily chosen by the kernel among the threads that are not blocking that signal.

- If a fatal signal is sent to a multithreaded application, the kernel will kill all threads of the application—not just the thread to which the signal has been delivered.

In order to comply with the POSIX standard, the Linux 2.6 kernel implements a multithreaded application as a set of lightweight processes belonging to the same thread group (see the section "Processes, Lightweight Processes, and Threads" in Chapter 3).

In this chapter the term "thread group" denotes any thread group, even if it is composed by a single (conventional) process. For instance, when we state that kill() can send a signal to a thread group, we imply that this system call can send a signal to a conventional process, too. We will use the term "process" to denote either a conventional process or a lightweight process—that is, a specific member of a thread group.

Furthermore, a pending signal is *private* if it has been sent to a specific process; it is *shared* if it has been sent to a whole thread group.

---

[*] There are two exceptions: it is not possible to send a signal to process 0 (*swapper*), and signals sent to process 1 (*init*) are always discarded unless they are caught. Therefore, process 0 never dies, while process 1 dies only when the *init* program terminates.

Data Structures Associated with Signals

For each process in the system, the kernel must keep track of what signals are currently pending or masked; the kernel must also keep track of how every thread group is supposed to handle every signal. To do this, the kernel uses several data structures accessible from the process descriptor. The most significant ones are shown in Figure 11-1.
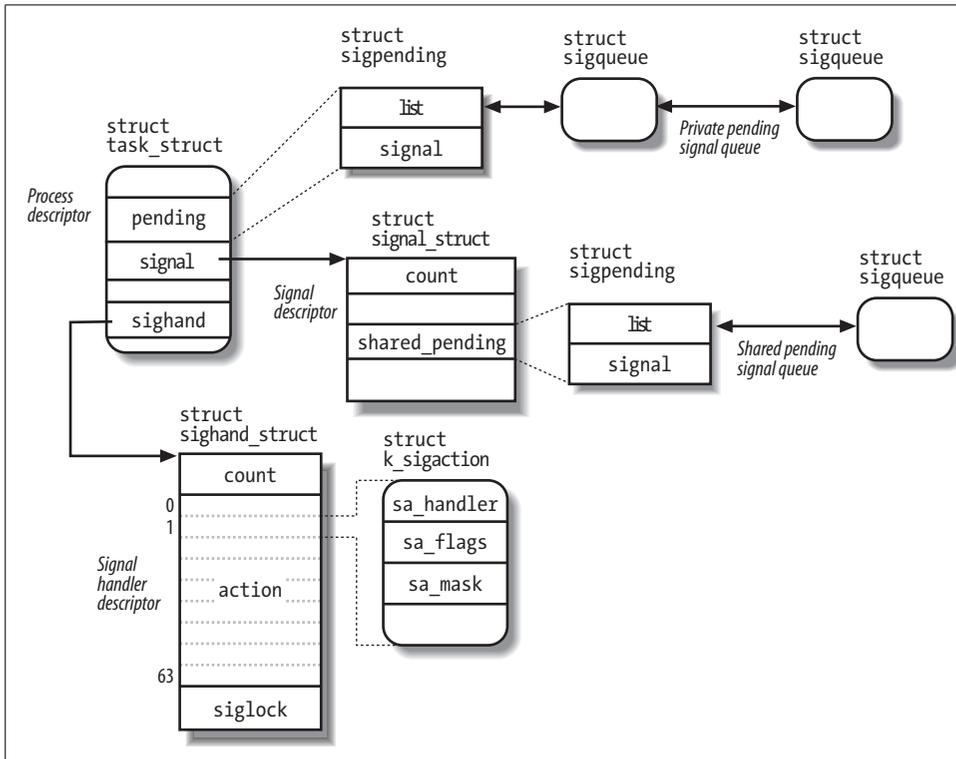
Figure 11-1. The most significant data structures related to signal handling

The fields of the process descriptor related to signal handling are listed in Table 11-3.

Table 11-3. Process descriptor fields related to signal handling

Type	Name	Description
struct signal_struct *	signal	Pointer to the process's signal descriptor
struct sighand_struct *	sighand	Pointer to the process's signal handler descriptor
sigset_t	blocked	Mask of blocked signals
sigset_t	real_blocked	Temporary mask of blocked signals (used by the rt_sigtimedwait( ) system call)
struct sigpending	pending	Data structure storing the private pending signals
unsigned long	sas_ss_sp	Address of alternative signal handler stack

Table 11-3. Process descriptor fields related to signal handling (continued)

Type	Name	Description
`size_t`	`sas_ss_size`	Size of alternative signal handler stack
`int (*) (void *)`	`notifier`	Pointer to a function used by a device driver to block some signals of the process
`void *`	`notifier_data`	Pointer to data that might be used by the notifier function (previous field of table)
`sigset_t *`	`notifier_mask`	Bit mask of signals blocked by a device driver through a notifier function

The `blocked` field stores the signals currently masked out by the process. It is a `sigset_t` array of bits, one for each signal type:

```
typedef struct {
    unsigned long sig[2];
} sigset_t;
```

Because each `unsigned long` number consists of 32 bits, the maximum number of signals that may be declared in Linux is 64 (the `_NSIG` macro specifies this value). No signal can have number 0, so the signal number corresponds to the index of the corresponding bit in a `sigset_t` variable plus one. Numbers between 1 and 31 correspond to the signals listed in Table 11-1, while numbers between 32 and 64 correspond to real-time signals.

The signal descriptor and the signal handler descriptor

The `signal` field of the process descriptor points to a *signal descriptor*, a `signal_struct` structure that keeps track of the shared pending signals. Actually, the signal descriptor also includes fields not strictly related to signal handling, such as the `rlim` per-process resource limit array (see the section "Process Resource Limits" in Chapter 3), or the `pgrp` and `session` fields, which store the PIDs of the group leader and of the session leader of the process, respectively (see the section "Relationships Among Processes" in Chapter 3). In fact, as mentioned in the section "The clone(), fork(), and vfork() System Calls" in Chapter 3, the signal descriptor is shared by all processes belonging to the same thread group—that is, all processes created by invoking the `clone()` system call with the `CLONE_THREAD` flag set—thus the signal descriptor includes the fields that must be identical for every process in the same thread group.

The fields of a signal descriptor somewhat related to signal handling are shown in Table 11-4.

Table 11-4. The fields of the signal descriptor related to signal handling

Type	Name	Description
`atomic_t`	`count`	Usage counter of the signal descriptor
`atomic_t`	`live`	Number of live processes in the thread group

Table 11-4. The fields of the signal descriptor related to signal handling (continued)

Type	Name	Description
wait_queue_head_t	wait_chldexit	Wait queue for the processes sleeping in a wait4() system call
struct task_struct *	curr_target	Descriptor of the last process in the thread group that received a signal
struct sigpending	shared_pending	Data structure storing the shared pending signals
int	group_exit_code	Process termination code for the thread group
struct task_struct *	group_exit_task	Used when killing a whole thread group
int	notify_count	Used when killing a whole thread group
int	group_stop_count	Used when stopping a whole thread group
unsigned int	flags	Flags used when delivering signals that modify the status of the process

Besides the signal descriptor, every process refers also to a *signal handler descriptor*, which is a sighand_struct structure describing how each signal must be handled by the thread group. Its fields are shown in Table 11-5.

Table 11-5. The fields of the signal handler descriptor

Type	Name	Description
atomic_t	count	Usage counter of the signal handler descriptor
struct k_sigaction [64]	action	Array of structures specifying the actions to be performed upon delivering the signals
spinlock_t	siglock	Spin lock protecting both the signal descriptor and the signal handler descriptor

As mentioned in the section "The clone(), fork(), and vfork() System Calls" in Chapter 3, the signal handler descriptor may be shared by several processes by invoking the clone() system call with the CLONE_SIGHAND flag set; the count field in this descriptor specifies the number of processes that share the structure. In a POSIX multithreaded application, all lightweight processes in the thread group refer to the same signal descriptor and to the same signal handler descriptor.

The sigaction data structure

Some architectures assign properties to a signal that are visible only to the kernel. Thus, the properties of a signal are stored in a k_sigaction structure, which contains both the properties hidden from the User Mode process and the more familiar sigaction structure that holds all the properties a User Mode process can see. Actually, on the 80×86 platform, all signal properties are visible to User Mode processes.

Thus the `k_sigaction` structure simply reduces to a single `sa` structure of type `sigaction`, which includes the following fields:*

`sa_handler`

> This field specifies the type of action to be performed; its value can be a pointer to the signal handler, `SIG_DFL` (that is, the value 0) to specify that the default action is performed, or `SIG_IGN` (that is, the value 1) to specify that the signal is ignored.

`sa_flags`

> This set of flags specifies how the signal must be handled; some of them are listed in Table 11-6.†

`sa_mask`

> This `sigset_t` variable specifies the signals to be masked when running the signal handler.

Table 11-6. Flags specifying how to handle a signal

Flag Name	Description
SA_NOCLDSTOP	Applies only to `SIGCHLD`; do not send `SIGCHLD` to the parent when the process is stopped
SA_NOCLDWAIT	Applies only to `SIGCHLD`; do not create a zombie when the process terminates
SA_SIGINFO	Provide additional information to the signal handler (see the later section "Changing a Signal Action")
SA_ONSTACK	Use an alternative stack for the signal handler (see the later section "Catching the Signal")
SA_RESTART	Interrupted system calls are automatically restarted (see the later section "Reexecution of System Calls")
SA_NODEFER, SA_NOMASK	Do not mask the signal while executing the signal handler
SA_RESETHAND, SA_ONESHOT	Reset to default action after executing the signal handler

The pending signal queues

As we have seen in Table 11-2 earlier in the chapter, there are several system calls that can generate a signal: some of them—`kill()` and `rt_sigqueueinfo()`—send a signal to a whole thread group, while others—`tkill()` and `tgkill()`—send a signal to a specific process.

---

* The `sigaction` structure used by User Mode applications to pass parameters to the `signal()` and `sigaction()` system calls is slightly different from the structure used by the kernel, although it stores essentially the same information.

† For historical reasons, these flags have the same prefix "SA_" as the flags of the `irqaction` descriptor (see Table 4-7 in Chapter 4); nevertheless there is no relation between the two sets of flags.

Thus, in order to keep track of what signals are currently pending, the kernel associates two pending signal queues to each process:

- The *shared pending signal queue*, rooted at the shared_pending field of the signal descriptor, stores the pending signals of the whole thread group.
- The *private pending signal queue*, rooted at the pending field of the process descriptor, stores the pending signals of the specific (lightweight) process.

A pending signal queue consists of a sigpending data structure, which is defined as follows:

```
struct sigpending {
    struct list_head list;
    sigset_t signal;
}
```

The signal field is a bit mask specifying the pending signals, while the list field is the head of a doubly linked list containing sigqueue data structures; the fields of this structure are shown in Table 11-7.

Table 11-7. The fields of the sigqueue data structure

Type	Name	Description
struct list_head	list	Links for the pending signal queue's list
spinlock_t *	lock	Pointer to the siglock field in the signal handler descriptor corresponding to the pending signal
int	flags	Flags of the sigqueue data structure
siginfo_t	info	Describes the event that raised the signal
struct user_struct *	user	Pointer to the per-user data structure of the process's owner (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3)

The siginfo_t data structure is a 128-byte data structure that stores information about an occurrence of a specific signal; it includes the following fields:

si_signo
> The signal number

si_errno
> The error code of the instruction that caused the signal to be raised, or 0 if there was no error

si_code
> A code identifying who raised the signal (see Table 11-8)

Table 11-8. The most significant signal sender codes

Code Name	Sender
SI_USER	kill( ) and raise( ) (see the later section "System Calls Related to Signal Handling")
SI_KERNEL	Generic kernel function
SI_QUEUE	sigqueue( ) (see the later section "System Calls Related to Signal Handling")

Table 11-8. The most significant signal sender codes (continued)

Code Name	Sender
SI_TIMER	Timer expiration
SI_ASYNCIO	Asynchronous I/O completion
SI_TKILL	tkill( ) and tgkill( ) (see the later section "System Calls Related to Signal Handling")

_sifields

> A union storing information depending on the type of signal. For instance, the siginfo_t data structure relative to an occurrence of the SIGKILL signal records the PID and the UID of the sender process here; conversely, the data structure relative to an occurrence of the SIGSEGV signal stores the memory address whose access caused the signal to be raised.

Operations on Signal Data Structures

Several functions and macros are used by the kernel to handle signals. In the following description, set is a pointer to a sigset_t variable, nsig is the number of a signal, and mask is an unsigned long bit mask.

sigemptyset(set) *and* sigfillset(set)

> Sets the bits in the sigset_t variable to 0 or 1, respectively.

sigaddset(set,nsig) *and* sigdelset(set,nsig)

> Sets the bit of the sigset_t variable corresponding to signal nsig to 1 or 0, respectively. In practice, sigaddset( ) reduces to:
>
>   set->sig[(nsig - 1) / 32] |= 1UL << ((nsig - 1) % 32);
>
> and sigdelset( ) to:
>
>   set->sig[(nsig - 1) / 32] &= ~(1UL << ((nsig - 1) % 32));

sigaddsetmask(set,mask) *and* sigdelsetmask(set,mask)

> Sets all the bits of the sigset_t variable whose corresponding bits of mask are on 1 or 0, respectively. They can be used only with signals that are between 1 and 32. The corresponding functions reduce to:
>
>   set->sig[0] |= mask;
>
> and to:
>
>   set->sig[0] &= ~mask;

sigismember(set,nsig)

> Returns the value of the bit of the sigset_t variable corresponding to the signal nsig. In practice, this function reduces to:
>
>   return 1 & (set->sig[(nsig-1) / 32] >> ((nsig-1) % 32));

sigmask(nsig)

> Yields the bit index of the signal nsig. In other words, if the kernel needs to set, clear, or test a bit in an element of sigset_t that corresponds to a particular signal, it can derive the proper bit through this macro.

sigandsets(d,s1,s2), sigorsets(d,s1,s2), *and* signandsets(d,s1,s2)
> Performs a logical AND, a logical OR, and a logical NAND, respectively, between the sigset_t variables to which s1 and s2 point; the result is stored in the sigset_t variable to which d points.

sigtestsetmask(set,mask)
> Returns the value 1 if any of the bits in the sigset_t variable that correspond to the bits set to 1 in mask is set; it returns 0 otherwise. It can be used only with signals that have a number between 1 and 32.

siginitset(set,mask)
> Initializes the low bits of the sigset_t variable corresponding to signals between 1 and 32 with the bits contained in mask, and clears the bits corresponding to signals between 33 and 63.

siginitsetinv(set,mask)
> Initializes the low bits of the sigset_t variable corresponding to signals between 1 and 32 with the complement of the bits contained in mask, and sets the bits corresponding to signals between 33 and 63.

signal_pending(p)
> Returns the value 1 (true) if the process identified by the *p process descriptor has nonblocked pending signals, and returns the value 0 (false) if it doesn't. The function is implemented as a simple check on the TIF_SIGPENDING flag of the process.

recalc_sigpending_tsk(t) *and* recalc_sigpending()
> The first function checks whether there are pending signals either for the process identified by the process descriptor at *t (by looking at the t->pending->signal field) or for the thread group to which the process belongs (by looking at the t->signal->shared_pending->signal field). The function then sets accordingly the TIF_SIGPENDING flag in t->thread_info->flags. The recalc_sigpending() function is equivalent to recalc_sigpending_tsk(current).

rm_from_queue(mask,q)
> Removes from the pending signal queue q the pending signals corresponding to the bit mask mask.

flush_sigqueue(q)
> Removes from the pending signal queue q all pending signals.

flush_signals(t)
> Deletes all signals sent to the process identified by the process descriptor at *t. This is done by clearing the TIF_SIGPENDING flag in t->thread_info->flags and invoking twice flush_sigqueue() on the t->pending and t->signal->shared_pending queues.

Generating a Signal

Many kernel functions generate signals: they accomplish the first phase of signal handling—described earlier in the section "The Role of Signals"—by updating one or more process descriptors as needed. They do not directly perform the second phase of delivering the signal but, depending on the type of signal and the state of the destination processes, may wake up some processes and force them to receive the signal.

When a signal is sent to a process, either from the kernel or from another process, the kernel generates it by invoking one of the functions listed in Table 11-9.

Table 11-9. Kernel functions that generate a signal for a process

Name	Description
send_sig()	Sends a signal to a single process
send_sig_info()	Like send_sig(), with extended information in a siginfo_t structure
force_sig()	Sends a signal that cannot be explicitly ignored or blocked by the process
force_sig_info()	Like force_sig(), with extended information in a siginfo_t structure
force_sig_specific()	Like force_sig(), but optimized for SIGSTOP and SIGKILL signals
sys_tkill()	System call handler of tkill() (see the later section "System Calls Related to Signal Handling")
sys_tgkill()	System call handler of tgkill()

All functions in Table 11-9 end up invoking the specific_send_sig_info() function described in the next section.

When a signal is sent to a whole thread group, either from the kernel or from another process, the kernel generates it by invoking one of the functions listed in Table 11-10.

Table 11-10. Kernel functions that generate a signal for a thread group

Name	Description
send_group_sig_info()	Sends a signal to a single thread group identified by the process descriptor of one of its members
kill_pg()	Sends a signal to all thread groups in a process group (see the section "Process Management" in Chapter 1)
kill_pg_info()	Like kill_pg(), with extended information in a siginfo_t structure
kill_proc()	Sends a signal to a single thread group identified by the PID of one of its members
kill_proc_info()	Like kill_proc(), with extended information in a siginfo_t structure
sys_kill()	System call handler of kill() (see the later section "System Calls Related to Signal Handling")
sys_rt_sigqueueinfo()	System call handler of rt_sigqueueinfo()

All functions in Table 11-10 end up invoking the `group_send_sig_info()` function, which is described in the later section "The group_send_sig_info() Function."

The specific_send_sig_info() Function

The `specific_send_sig_info()` function sends a signal to a specific process. It acts on three parameters:

sig

    The signal number.

info

    Either the address of a `siginfo_t` table or one of three special values: 0 means that the signal has been sent by a User Mode process, 1 means that it has been sent by the kernel, and 2 means that is has been sent by the kernel and the signal is `SIGSTOP` or `SIGKILL`.

t

    A pointer to the descriptor of the destination process.

The `specific_send_sig_info()` function must be invoked with local interrupts disabled and the `t->sighand->siglock` spin lock already acquired. The function executes the following steps:

1. Checks whether the process ignores the signal; in the affirmative case, returns 0 (signal not generated). The signal is ignored when all three conditions for ignoring a signal are satisfied, that is:

    • The process is not being traced (`PT_PTRACED` flag in `t->ptrace` clear).

    • The signal is not blocked (`sigismember(&t->blocked, sig)` returns 0).

    • The signal is either explicitly ignored (the `sa_handler` field of `t->sighand->action[sig-1]` is equal to `SIG_IGN`) or implicitly ignored (the `sa_handler` field is equal to `SIG_DFL` and the signal is `SIGCONT`, `SIGCHLD`, `SIGWINCH`, or `SIGURG`).

2. Checks whether the signal is non-real-time (`sig<32`) and another occurrence of the same signal is already pending in the private pending signal queue of the process (`sigismember(&t->pending.signal,sig)` returns 1): in the affirmative case, nothing has to be done, thus returns 0.

3. Invokes `send_signal(sig, info, t, &t->pending)` to add the signal to the set of pending signals of the process; this function is described in detail in the next section.

4. If `send_signal()` successfully terminated and the signal is not blocked (`sigismember(&t->blocked,sig)` returns 0), invokes the `signal_wake_up()` function to notify the process about the new pending signal. In turn, this function executes the following steps:

    a. Sets the `TIF_SIGPENDING` flags in `t->thread_info->flags`.

b. Invokes `try_to_wake_up()`—see the section "The try_to_wake_up() Function" in Chapter 7—to awake the process if it is either in `TASK_INTERRUPTIBLE` state, or in `TASK_STOPPED` state and the signal is `SIGKILL`.

c. If `try_to_wake_up()` returned 0, the process was already runnable: if so, it checks whether the process is already running on another CPU and, in this case, sends an interprocessor interrupt to that CPU to force a reschedule of the current process (see the section "Interprocessor Interrupt Handling" in Chapter 4). Because each process checks the existence of pending signals when returning from the `schedule()` function, the interprocessor interrupt ensures that the destination process quickly notices the new pending signal.

5. Returns 1 (the signal has been successfully generated).

The send_signal() Function

The `send_signal()` function inserts a new item in a pending signal queue. It receives as its parameters the signal number `sig`, the address `info` of a `siginfo_t` data structure (or a special code, see the description of `specific_send_sig_info()` in the previous section), the address `t` of the descriptor of the target process, and the address `signals` of the pending signal queue.

The function executes the following steps:

1. If the value of `info` is 2, the signal is either `SIGKILL` or `SIGSTOP` and it has been generated by the kernel via the `force_sig_specific()` function: in this case, it jumps to step 9. The action corresponding to these signals is immediately enforced by the kernel, thus the function may skip adding the signal to the pending signal queue.

2. If the number of pending signals of the process's owner (`t->user->sigpending`) is smaller than the current process's resource limit (`t->signal->rlim[RLIMIT_SIGPENDING].rlim_cur`), the function allocates a `sigqueue` data structure for the new occurrence of the signal:

       q = kmem_cache_alloc(sigqueue_cachep, GFP_ATOMIC);

3. If the number of pending signals of the process's owner is too high or the memory allocation in the previous step failed, it jumps to step 9.

4. Increases the number of pending signals of the owner (`t->user->sigpending`) and the reference counter of the per-user data structure pointed to by `t->user`.

5. Adds the `sigqueue` data structure in the pending signal queue `signals`:

       list_add_tail(&q->list, &signals->list);

6. Fills the `siginfo_t` table inside the `sigqueue` data structure:

       if ((unsigned long)info == 0) {
           q->info.si_signo = sig;
           q->info.si_errno = 0;
           q->info.si_code = SI_USER;

```
    q->info._sifields._kill._pid = current->pid;
    q->info._sifields._kill._uid = current->uid;
} else if ((unsigned long)info == 1) {
    q->info.si_signo = sig;
    q->info.si_errno = 0;
    q->info.si_code = SI_KERNEL;
    q->info._sifields._kill._pid = 0;
    q->info._sifields._kill._uid = 0;
} else
    copy_siginfo(&q->info, info);
```

The copy_siginfo() function copies the siginfo_t table passed by the caller.

7. Sets the bit corresponding to the signal in the bit mask of the queue:

   ```
   sigaddset(&signals->signal, sig);
   ```

8. Returns 0: the signal has been successfully appended to the pending signal queue.

9. Here, an item will not be added to the signal pending queue, because there are already too many pending signals, or there is no free memory for the sigqueue data structure, or the signal is immediately enforced by the kernel. If the signal is real-time and was sent through a kernel function that is explicitly required to queue it, the function returns the error code -EAGAIN:

   ```
   if (sig>=32 && info && (unsigned long) info != 1 &&
       info->si_code != SI_USER)
       return -EAGAIN;
   ```

10. Sets the bit corresponding to the signal in the bit mask of the queue:

    ```
    sigaddset(&signals->signal, sig);
    ```

11. Returns 0: even if the signal has not been appended to the queue, the corresponding bit has been set in the bit mask of pending signals.

It is important to let the destination process receive the signal even if there is no room for the corresponding item in the pending signal queue. Suppose, for instance, that a process is consuming too much memory. The kernel must ensure that the kill() system call succeeds even if there is no free memory; otherwise, the system administrator doesn't have any chance to recover the system by terminating the offending process.

The group_send_sig_info() Function

The group_send_sig_info() function sends a signal to a whole thread group. It acts on three parameters: a signal number sig, the address info of a siginfo_t table—or alternatively the special values 0, 1, or 2, as explained in the earlier section "The specific_send_sig_info() Function"—and the address p of a process descriptor.

The function essentially executes the following steps:

1. Checks that the parameter sig is correct:

   ```
   if (sig < 0 || sig > 64)
       return -EINVAL;
   ```

2. If the signal is being sent by a User Mode process, it checks whether the operation is allowed. The signal is delivered only if at least one of the following conditions holds:

- The owner of the sending process has the proper capability (usually, this simply means the signal was issued by the system administrator; see Chapter 20).

- The signal is SIGCONT and the destination process is in the same login session of the sending process.

- Both processes belong to the same user.

If the User Mode process is not allowed to send the signal, the function returns the value -EPERM.

3. If the sig parameter has the value 0, it returns immediately without generating any signal:

```
if (!sig || !p->sighand)
    return 0;
```

Because 0 is not a valid signal number, it is used to allow the sending process to check whether it has the required privileges to send a signal to the destination thread group. The function also returns if the destination process is being killed, indicated by checking whether its signal handler descriptor has been released.

4. Acquires the p->sighand->siglock spin lock and disables local interrupts.

5. Invokes the handle_stop_signal() function, which checks for some types of signals that might nullify other pending signals for the destination thread group. The latter function executes the following steps:

   a. If the thread group is being killed (SIGNAL_GROUP_EXIT flag in the flags field of the signal descriptor set), it returns.

   b. If sig is a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal, the function invokes the rm_from_queue() function to remove the SIGCONT signal from the shared pending signal queue p->signal->shared_pending and from the private queues of all members of the thread group.

   c. If sig is SIGCONT, it invokes the rm_from_queue() function to remove any SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU signal from the shared pending signal queue p->signal->shared_pending; then, removes the same signals from the private pending signal queues of the processes belonging to the thread group, and awakens them:

   ```
   rm_from_queue(0x003c0000, &p->signal->shared_pending);
   t = p;
   do {
       rm_from_queue(0x003c0000, &t->pending);
       try_to_wake_up(t, TASK_STOPPED, 0);
       t = next_thread(t);
   } while (t != p);
   ```

The mask 0x003c0000 selects the four stop signals. At each iteration, the next_thread macro returns the descriptor address of a different lightweight process in the thread group (see the section "Relationships Among Processes" in Chapter 3).*

6. Checks whether the thread group ignores the signal; if so, returns the value 0 (success). The signal is ignored when all three conditions for ignoring a signal that are mentioned in the earlier section "The Role of Signals" are satisfied (see also step 1 in the earlier section "The specific_send_sig_info() Function").

7. Checks whether the signal is non-real-time and another occurrence of the same signal is already pending in the shared pending signal queue of the thread group: if so, nothing has to be done, thus returns the value 0 (success):

```
if (sig<32 && sigismember(&p->signal->shared_pending.signal,sig))
    return 0;
```

8. Invokes send_signal() to append the signal to the shared pending signal queue (see the previous section "The send_signal() Function"). If send_signal() returns a nonzero error code, it terminates while returning the same value.

9. Invokes the __group_complete_signal() function to wake up one lightweight process in the thread group (see below).

10. Releases the p->sighand->siglock spin lock and enables local interrupts.

11. Returns 0 (success).

The __group_complete_signal() function scans the processes in the thread group, looking for a process that can receive the new signal. A process may be selected if it satisfies all the following conditions:

- The process does not block the signal.
- The process is not in state EXIT_ZOMBIE, EXIT_DEAD, TASK_TRACED, or TASK_STOPPED (as an exception, the process can be in the TASK_TRACED or TASK_STOPPED states if the signal is SIGKILL).
- The process is not being killed—that is, its PF_EXITING flag is not set.
- Either the process is currently in execution on a CPU, or its TIF_SIGPENDING flag is not already set. (In fact, there is no point in awakening a process that has pending signals: in general, this operation has been already performed by the kernel control path that set the TIF_SIGPENDING flag. On the other hand, if a process is currently in execution, it should be notified of the new pending signal.)

A thread group might include many processes that satisfy the above conditions. The function selects one of them as follows:

---

* The actual code is more complicated than the fragment just shown, because handle_stop_signal() also takes care of the unusual case of the SIGCONT signal being caught, as well as of the race conditions due to a SIGCONT signal occurring while all processes in the thread group are being stopped.

- If the process identified by p—the descriptor address passed as parameter of the `group_send_sig_info()` function—satisfies all the prior rules and can thus receive the signal, the function selects it.

- Otherwise, the function searches for a suitable process by scanning the members of the thread group, starting from the process that received the last thread group's signal (`p->signal->curr_target`).

If `__group_complete_signal()` succeeds in finding a suitable process, it sets up the delivery of the signal to the selected process. First, the function checks whether the signal is fatal: in this case, the whole thread group is killed by sending `SIGKILL` signals to each lightweight process in the group. Otherwise, if the signal is not fatal, the function invokes the `signal_wake_up()` function to notify the selected process that it has a new pending signal (see step 4 in the earlier section "The specific_send_sig_info() Function").

Delivering a Signal

We assume that the kernel noticed the arrival of a signal and invoked one of the functions mentioned in the previous sections to prepare the process descriptor of the process that is supposed to receive the signal. But in case that process was not running on the CPU at that moment, the kernel deferred the task of delivering the signal. We now turn to the activities that the kernel performs to ensure that pending signals of a process are handled.

As mentioned in the section "Returning from Interrupts and Exceptions" in Chapter 4, the kernel checks the value of the `TIF_SIGPENDING` flag of the process before allowing the process to resume its execution in User Mode. Thus, the kernel checks for the existence of pending signals every time it finishes handling an interrupt or an exception.

To handle the nonblocked pending signals, the kernel invokes the `do_signal()` function, which receives two parameters:

regs
    The address of the stack area where the User Mode register contents of the current process are saved.

oldset
    The address of a variable where the function is supposed to save the bit mask array of blocked signals. It is `NULL` if there is no need to save the bit mask array.

Our description of the `do_signal()` function will focus on the general mechanism of signal delivery; the actual code is burdened with lots of details dealing with race conditions and other special cases—such as freezing the system, generating core dumps, stopping and killing a whole thread group, and so on. We will quietly skip all these details.

As already mentioned, the do_signal() function is usually only invoked when the CPU is going to return in User Mode. For that reason, if an interrupt handler invokes do_signal(), the function simply returns:

```
if ((regs->xcs & 3) != 3)
    return 1;
```

If the oldset parameter is NULL, the function initializes it with the address of the current->blocked field:

```
if (!oldset)
    oldset = &current->blocked;
```

The heart of the do_signal() function consists of a loop that repeatedly invokes the dequeue_signal() function until no nonblocked pending signals are left in both the private and shared pending signal queues. The return code of dequeue_signal() is stored in the signr local variable. If its value is 0, it means that all pending signals have been handled and do_signal() can finish. As long as a nonzero value is returned, a pending signal is waiting to be handled. dequeue_signal() is invoked again after do_signal() handles the current signal.

The dequeue_signal() considers first all signals in the private pending signal queue, starting from the lowest-numbered signal, then the signals in the shared queue. It updates the data structures to indicate that the signal is no longer pending and returns its number. This task involves clearing the corresponding bit in current->pending.signal or current->signal->shared_pending.signal, and invoking recalc_sigpending() to update the value of the TIF_SIGPENDING flag.

Let's see how the do_signal() function handles each pending signal whose number is returned by dequeue_signal(). First, it checks whether the current receiver process is being monitored by some other process; in this case, do_signal() invokes do_notify_parent_cldstop() and schedule() to make the monitoring process aware of the signal handling.

Then do_signal() loads the ka local variable with the address of the k_sigaction data structure of the signal to be handled:

```
ka = &current->sig->action[signr-1];
```

Depending on the contents, three kinds of actions may be performed: ignoring the signal, executing a default action, or executing a signal handler.

When a delivered signal is explicitly ignored, the do_signal() function simply continues with a new execution of the loop and therefore considers another pending signal:

```
if (ka->sa.sa_handler == SIG_IGN)
    continue;
```

In the following two sections we will describe how a default action and a signal handler are executed.

Executing the Default Action for the Signal

If `ka->sa.sa_handler` is equal to `SIG_DFL`, `do_signal()` must perform the default action of the signal. The only exception comes when the receiving process is *init*, in which case the signal is discarded as described in the earlier section "Actions Performed upon Delivering a Signal":

```
if (current->pid == 1)
    continue;
```

For other processes, the signals whose default action is "ignore" are also easily handled:

```
if (signr==SIGCONT || signr==SIGCHLD ||
        signr==SIGWINCH || signr==SIGURG)
    continue;
```

The signals whose default action is "stop" may stop all processes in the thread group. To do this, `do_signal()` sets their states to `TASK_STOPPED` and then invokes the `schedule()` function (see the section "The schedule() Function" in Chapter 7):

```
if (signr==SIGSTOP || signr==SIGTSTP ||
        signr==SIGTTIN || signr==SIGTTOU) {
    if (signr != SIGSTOP &&
            is_orphaned_pgrp(current->signal->pgrp))
        continue;
    do_signal_stop(signr);
}
```

The difference between `SIGSTOP` and the other signals is subtle: `SIGSTOP` always stops the thread group, while the other signals stop the thread group only if it is not in an "orphaned process group." The POSIX standard specifies that a process group is *not* orphaned as long as there is a process in the group that has a parent in a different process group but in the same session. Thus, if the parent process dies but the user who started the process is still logged in, the process group is not orphaned.

The `do_signal_stop()` function checks whether current is the first process being stopped in the thread group. If so, it activates a "group stop": essentially, the function sets the `group_stop_count` field in the signal descriptor to a positive value, and awakens each process in the thread group. Each such process, in turn, looks at this field to recognize that a group stop is in progress, changes its state to `TASK_STOPPED`, and invokes `schedule()`. The `do_signal_stop()` function also sends a `SIGCHLD` signal to the parent process of the thread group leader, unless the parent has set the `SA_NOCLDSTOP` flag of `SIGCHLD`.

The signals whose default action is "dump" may create a core file in the process working directory; this file lists the complete contents of the process's address space and CPU registers. After `do_signal()` creates the core file, it kills the thread group. The default action of the remaining 18 signals is "terminate," which consists of simply killing the thread group. To kill the whole thread group, the function invokes

do_group_exit(), which executes a clean "group exit" procedure (see the section "Process Termination" in Chapter 3).

Catching the Signal

If a handler has been established for the signal, the do_signal() function must enforce its execution. It does this by invoking handle_signal():

```
handle_signal(signr, &info, &ka, oldset, regs);
if (ka->sa.sa_flags & SA_ONESHOT)
    ka->sa.sa_handler = SIG_DFL;
return 1;
```

If the received signal has the SA_ONESHOT flag set, it must be reset to its default action, so that further occurrences of the same signal will not trigger again the execution of the signal handler. Notice how do_signal() returns after having handled a single signal. Other pending signals won't be considered until the next invocation of do_signal(). This approach ensures that real-time signals will be dealt with in the proper order.

Executing a signal handler is a rather complex task because of the need to juggle stacks carefully while switching between User Mode and Kernel Mode. We explain exactly what is entailed here:

Signal handlers are functions defined by User Mode processes and included in the User Mode code segment. The handle_signal() function runs in Kernel Mode while signal handlers run in User Mode; this means that the current process must first execute the signal handler in User Mode before being allowed to resume its "normal" execution. Moreover, when the kernel attempts to resume the normal execution of the process, the Kernel Mode stack no longer contains the hardware context of the interrupted program, because the Kernel Mode stack is emptied at every transition from User Mode to Kernel Mode.

An additional complication is that signal handlers may invoke system calls. In this case, after the service routine executes, control must be returned to the signal handler instead of to the normal flow of code of the interrupted program.

The solution adopted in Linux consists of copying the hardware context saved in the Kernel Mode stack onto the User Mode stack of the current process. The User Mode stack is also modified in such a way that, when the signal handler terminates, the sigreturn() system call is automatically invoked to copy the hardware context back on the Kernel Mode stack and to restore the original content of the User Mode stack.

Figure 11-2 illustrates the flow of execution of the functions involved in catching a signal. A nonblocked signal is sent to a process. When an interrupt or exception occurs, the process switches into Kernel Mode. Right before returning to User Mode, the kernel executes the do_signal() function, which in turn handles the signal (by invoking handle_signal()) and sets up the User Mode stack (by invoking setup_frame() or setup_rt_frame()). When the process switches again to User Mode, it

starts executing the signal handler, because the handler's starting address was forced into the program counter. When that function terminates, the return code placed on the User Mode stack by the setup_frame( ) or setup_rt_frame( ) function is executed. This code invokes the sigreturn( ) or the rt_sigreturn( ) system call; the corresponding service routines copy the hardware context of the normal program to the Kernel Mode stack and restore the User Mode stack back to its original state (by invoking restore_sigcontext( )). When the system call terminates, the normal program can thus resume its execution.
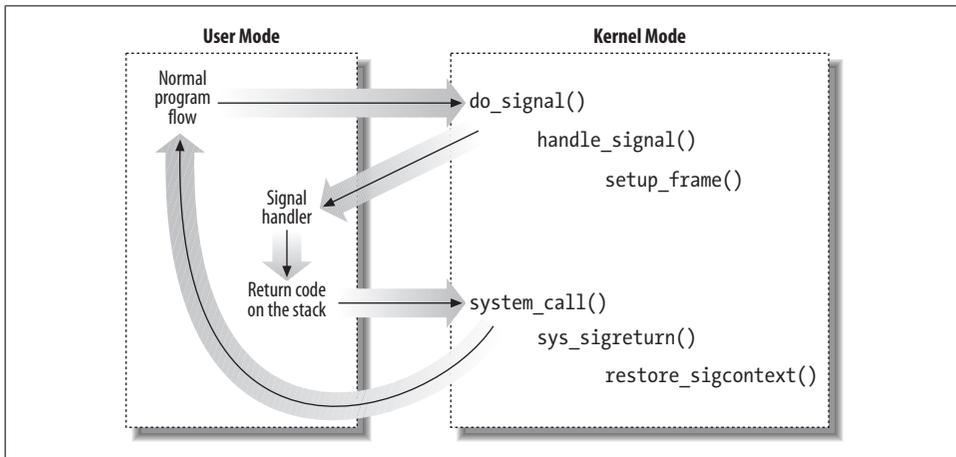
Figure 11-2. Catching a signal

Let's now examine in detail how this scheme is carried out.

Setting up the frame

To properly set the User Mode stack of the process, the handle_signal( ) function invokes either setup_frame( ) (for signals that do not require a siginfo_t table; see the section "System Calls Related to Signal Handling" later in this chapter) or setup_rt_frame( ) (for signals that do require a siginfo_t table). To choose among these two functions, the kernel checks the value of the SA_SIGINFO flag in the sa_flags field of the sigaction table associated with the signal.

The setup_frame( ) function receives four parameters, which have the following meanings:

sig
    Signal number

ka
    Address of the k_sigaction table associated with the signal

oldset
    Address of a bit mask array of blocked signals

regs
:   Address in the Kernel Mode stack area where the User Mode register contents are saved

The setup_frame() function pushes onto the User Mode stack a data structure called a *frame*, which contains the information needed to handle the signal and to ensure the correct return to the sys_sigreturn() function. A frame is a sigframe table that includes the following fields (see Figure 11-3):

pretcode
:   Return address of the signal handler function; it points to the code at the __kernel_sigreturn label (see below).

sig
:   The signal number; this is the parameter required by the signal handler.

sc

:   Structure of type sigcontext containing the hardware context of the User Mode process right before switching to Kernel Mode (this information is copied from the Kernel Mode stack of current). It also contains a bit array that specifies the blocked regular signals of the process.

fpstate
:   Structure of type _fpstate that may be used to store the floating point registers of the User Mode process (see the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3).

extramask
:   Bit array that specifies the blocked real-time signals.

retcode
:   8-byte code issuing a sigreturn() system call. In earlier versions of Linux, this code was effectively executed to return from the signal handler; in Linux 2.6, however, it is used only as a signature, so that debuggers can recognize the signal stack frame.
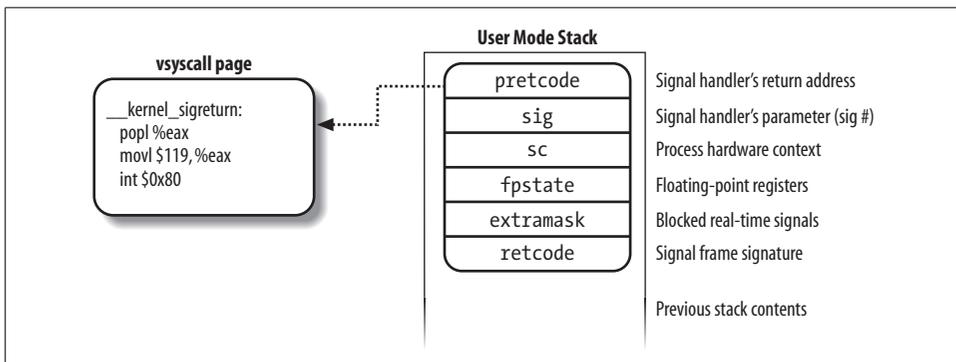
Figure 11-3. Frame on the User Mode stack

The setup_frame() function starts by invoking get_sigframe() to compute the first memory location of the frame. That memory location is usually* in the User Mode stack, so the function returns the value:

```
(regs->esp - sizeof(struct sigframe)) & 0xfffffff8
```

Because stacks grow toward lower addresses, the initial address of the frame is obtained by subtracting its size from the address of the current stack top and aligning the result to a multiple of 8.

The returned address is then verified by means of the access_ok macro; if it is valid, the function repeatedly invokes __put_user() to fill all the fields of the frame. The pretcode field in the frame is initialized to &__kernel_sigreturn, the address of some glue code placed in the vsyscall page (see the section "Issuing a System Call via the sysenter Instruction" in Chapter 10).

Once this is done, the function modifies the regs area of the Kernel Mode stack, thus ensuring that control is transferred to the signal handler when current resumes its execution in User Mode:

```
regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
regs->eax = (unsigned long) sig;
regs->edx = regs->ecx = 0;
regs->xds = regs->xes = regs->xss = __USER_DS;
regs->xcs = __USER_CS;
```

The setup_frame() function terminates by resetting the segmentation registers saved on the Kernel Mode stack to their default value. Now the information needed by the signal handler is on the top of the User Mode stack.

The setup_rt_frame() function is similar to setup_frame(), but it puts on the User Mode stack an *extended frame* (stored in the rt_sigframe data structure) that also includes the content of the siginfo_t table associated with the signal. Moreover, this function sets the pretcode field so that it points to the __kernel_rt_sigreturn code in the vsyscall page.

Evaluating the signal flags

After setting up the User Mode stack, the handle_signal() function checks the values of the flags associated with the signal. If the signal does not have the SA_NODEFER flag set, the signals in the sa_mask field of the sigaction table must be blocked during the execution of the signal handler:

```
if (!(ka->sa.sa_flags & SA_NODEFER)) {
    spin_lock_irq(&current->sighand->siglock);
```

---

* Linux allows processes to specify an alternative stack for their signal handlers by invoking the signaltstack( ) system call; this feature is also required by the X/Open standard. When an alternative stack is present, the get_sigframe( ) function returns an address inside that stack. We don't discuss this feature further, because it is conceptually similar to regular signal handling.

```
        sigorsets(&current->blocked, &current->blocked, &ka->sa.sa_mask);
        sigaddset(&current->blocked, sig);
        recalc_sigpending(current);
        spin_unlock_irq(&current->sighand->siglock);
    }
```

As described earlier, the `recalc_sigpending()` function checks whether the process has nonblocked pending signals and sets its `TIF_SIGPENDING` flag accordingly.

The function returns then to `do_signal()`, which also returns immediately.

Starting the signal handler

When `do_signal()` returns, the current process resumes its execution in User Mode. Because of the preparation by `setup_frame()` described earlier, the `eip` register points to the first instruction of the signal handler, while `esp` points to the first memory location of the frame that has been pushed on top of the User Mode stack. As a result, the signal handler is executed.

Terminating the signal handler

When the signal handler terminates, the return address on top of the stack points to the code in the vsyscall page referenced by the `pretcode` field of the frame:

```
__kernel_sigreturn:
  popl %eax
  movl $__NR_sigreturn, %eax
  int $0x80
```

Therefore, the signal number (that is, the `sig` field of the frame) is discarded from the stack; the `sigreturn()` system call is then invoked.

The `sys_sigreturn()` function computes the address of the `pt_regs` data structure regs, which contains the hardware context of the User Mode process (see the section "Parameter Passing" in Chapter 10). From the value stored in the `esp` field, it can thus derive and check the frame address inside the User Mode stack:

```
frame = (struct sigframe *)(regs.esp - 8);
if (verify_area(VERIFY_READ, frame, sizeof(*frame)) {
    force_sig(SIGSEGV, current);
    return 0;
}
```

Then the function copies the bit array of signals that were blocked before invoking the signal handler from the `sc` field of the frame to the `blocked` field of current. As a result, all signals that have been masked for the execution of the signal handler are unblocked. The `recalc_sigpending()` function is then invoked.

The `sys_sigreturn()` function must at this point copy the process hardware context from the `sc` field of the frame to the Kernel Mode stack and remove the frame from

the User Mode stack; it performs these two tasks by invoking the restore_
sigcontext( ) function.

If the signal was sent by a system call such as rt_sigqueueinfo( ) that required a
siginfo_t table to be associated with the signal, the mechanism is similar. The
pretcode field of the extended frame points to the __kernel_rt_sigreturn code in the
vsyscall page, which in turn invokes the rt_sigreturn( ) system call; the correspond-
ing sys_rt_sigreturn( ) service routine copies the process hardware context from the
extended frame to the Kernel Mode stack and restores the original User Mode stack
content by removing the extended frame from it.

Reexecution of System Calls

The request associated with a system call cannot always be immediately satisfied by
the kernel; when this happens, the process that issued the system call is put in a
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE state.

If the process is put in a TASK_INTERRUPTIBLE state and some other process sends a sig-
nal to it, the kernel puts it in the TASK_RUNNING state without completing the system
call (see the section "Returning from Interrupts and Exceptions" in Chapter 4). The
signal is delivered to the process while switching back to User Mode. When this hap-
pens, the system call service routine does not complete its job, but returns an EINTR,
ERESTARTNOHAND, ERESTART_RESTARTBLOCK, ERESTARTSYS, or ERESTARTNOINTR error code.

In practice, the only error code a User Mode process can get in this situation is EINTR,
which means that the system call has not been completed. (The application program-
mer may check this code and decide whether to reissue the system call.) The remain-
ing error codes are used internally by the kernel to specify whether the system call
may be reexecuted automatically after the signal handler termination.

Table 11-11 lists the error codes related to unfinished system calls and their impact
for each of the three possible signal actions. The terms that appear in the entries are
defined in the following list:

Terminate
>   The system call will not be automatically reexecuted; the process will resume its
>   execution in User Mode at the instruction following the int $0x80 or sysenter
>   one and the eax register will contain the -EINTR value.

Reexecute
>   The kernel forces the User Mode process to reload the eax register with the sys-
>   tem call number and to reexecute the int $0x80 or sysenter instruction; the pro-
>   cess is not aware of the reexecution and the error code is not passed to it.

Depends
>   The system call is reexecuted only if the SA_RESTART flag of the delivered signal is
>   set; otherwise, the system call terminates with a -EINTR error code.

Table 11-11. Reexecution of system calls

Signal Action	Error codes and their impact on system call execution			
	EINTR	ERESTARTSYS	ERESTARTNOHAND ERESTART_RESTARTBLOCK[a]	ERESTARTNOINTR
Default	Terminate	Reexecute	Reexecute	Reexecute
Ignore	Terminate	Reexecute	Reexecute	Reexecute
Catch	Terminate	Depends	Terminate	Reexecute

a The ERESTARTNOHAND and ERESTART_RESTARTBLOCK error codes differ on the mechanism used to restart the system call (see below).

When delivering a signal, the kernel must be sure that the process really issued a system call before attempting to reexecute it. This is where the orig_eax field of the regs hardware context plays a critical role. Let's recall how this field is initialized when the interrupt or exception handler starts:

Interrupt

The field contains the IRQ number associated with the interrupt minus 256 (see the section "Saving the registers for the interrupt handler" in Chapter 4).

0x80 *exception (also* sysenter*)*

The field contains the system call number (see the section "Entering and Exiting a System Call" in Chapter 10).

Other exceptions

The field contains the value –1 (see the section "Saving the Registers for the Exception Handler" in Chapter 4).

Therefore, a nonnegative value in the orig_eax field means that the signal has woken up a TASK_INTERRUPTIBLE process that was sleeping in a system call. The service routine recognizes that the system call was interrupted, and thus returns one of the previously mentioned error codes.

Restarting a system call interrupted by a non-caught signal

If the signal is explicitly ignored or if its default action is enforced, do_signal( ) analyzes the error code of the system call to decide whether the unfinished system call must be automatically reexecuted, as specified in Table 11-11. If the call must be restarted, the function modifies the regs hardware context so that, when the process is back in User Mode, eip points either to the int $0x80 instruction or to the sysenter instruction, and eax contains the system call number:

```
if (regs->orig_eax >= 0) {
    if (regs->eax == -ERESTARTNOHAND || regs->eax == -ERESTARTSYS ||
        regs->eax == -ERESTARTNOINTR) {
        regs->eax = regs->orig_eax;
        regs->eip -= 2;
    }
    if (regs->eax == -ERESTART_RESTARTBLOCK) {
```

```
        regs->eax = __NR_restart_syscall;
        regs->eip -= 2;
    }
}
```

The `regs->eax` field is filled with the return code of a system call service routine (see the section "Entering and Exiting a System Call" in Chapter 10). Notice that both the `int $0x80` and `sysreturn` instructions are two bytes long so the function subtracts 2 from `eip` in order to set it to the instruction that triggers the system call.

The error code `ERESTART_RESTARTBLOCK` is special, because the eax register is set to the number of the `restart_syscall()` system call; thus, the User Mode process does not restart the same system call that was interrupted by the signal. This error code is only used by time-related system calls that, when restarted, should adjust their User Mode parameters. A typical example is the `nanosleep()` system call (see the section "An Application of Dynamic Timers: the nanosleep() System Call" in Chapter 6): suppose that a process invokes it to pause the execution for 20 milliseconds, and that a signal occurs 10 milliseconds later. If the system call would be restarted as usual, the total delay time would exceed 30 milliseconds.

Instead, the service routine of the `nanosleep()` system call fills the `restart_block` field in the `current`'s `thread_info` structure with the address of a special service routine to be used when restarting, and returns -`ERESTART_RESTARTBLOCK` if interrupted. The `sys_restart_syscall()` service routine just executes the special `nanosleep()`'s service routine, which adjusts the delay to consider the time elapsed between the invocation of the original system call and its restarting.

Restarting a system call for a caught signal

If the signal is caught, `handle_signal()` analyzes the error code and, possibly, the `SA_RESTART` flag of the `sigaction` table to decide whether the unfinished system call must be reexecuted:

```
if (regs->orig_eax >= 0) {
    switch (regs->eax) {
        case -ERESTART_RESTARTBLOCK:
        case -ERESTARTNOHAND:
            regs->eax = -EINTR;
            break;
        case -ERESTARTSYS:
            if (!(ka->sa.sa_flags & SA_RESTART)) {
                regs->eax = -EINTR;
                break;
            }
        /* fallthrough */
        case -ERESTARTNOINTR:
            regs->eax = regs->orig_eax;
            regs->eip -= 2;
    }
}
```

If the system call must be restarted, `handle_signal()` proceeds exactly as `do_signal()`; otherwise, it returns an `-EINTR` error code to the User Mode process.

System Calls Related to Signal Handling

As stated in the introduction of this chapter, programs running in User Mode are allowed to send and receive signals. This means that a set of system calls must be defined to allow these kinds of operations. Unfortunately, for historical reasons, several system calls exist that serve essentially the same purpose. As a result, some of these system calls are never invoked. For instance, `sys_sigaction()` and `sys_rt_sigaction()` are almost identical, so the `sigaction()` wrapper function included in the C library ends up invoking `sys_rt_sigaction()` instead of `sys_sigaction()`. We will describe some of the most significant system calls in the following sections.

The kill( ) System Call

The `kill(pid,sig)` system call is commonly used to send signals to conventional processes or multithreaded applications; its corresponding service routine is the `sys_kill()` function. The integer `pid` parameter has several meanings, depending on its numerical value:

pid > 0

    The `sig` signal is sent to the thread group of the process whose PID is equal to `pid`.

pid = 0

    The `sig` signal is sent to all thread groups of the processes in the same process group as the calling process.

pid = –1

    The signal is sent to all processes, except *swapper* (PID 0), *init* (PID 1), and `current`.

pid < –1

    The signal is sent to all thread groups of the processes in the process group *–pid*.

The `sys_kill()` function sets up a minimal `siginfo_t` table for the signal, and then invokes `kill_something_info()`:

```
info.si_signo = sig;
info.si_errno = 0;
info.si_code = SI_USER;
info._sifields._kill._pid = current->tgid;
info._sifields._kill._uid = current->uid;
return kill_something_info(sig, &info, pid);
```

The `kill_something_info()` function, in turn, invokes either `kill_proc_info()` (to send the signal to a single thread group via `group_send_sig_info()`), or `kill_pg_info()` (to scan all processes in the destination process group and invoke `send_sig_info()` for

each of them), or repeatedly `group_send_sig_info()` for each process in the system (if pid is -1).

The `kill()` system call is able to send every signal, even the so-called real-time signals that have numbers ranging from 32 to 64. However, as we saw in the earlier section "Generating a Signal," the `kill()` system call does not ensure that a new element is added to the pending signal queue of the destination process, so multiple instances of pending signals can be lost. Real-time signals should be sent by means of a system call such as `rt_sigqueueinfo()` (see the later section "System Calls for Real-Time Signals").

System V and BSD Unix variants also have a `killpg()` system call, which is able to explicitly send a signal to a group of processes. In Linux, the function is implemented as a library function that uses the `kill()` system call. Another variation is `raise()`, which sends a signal to the current process (that is, to the process executing the function). In Linux, `raise()` is implemented as a library function.

The tkill() and tgkill() System Calls

The `tkill()` and `tgkill()` system calls send a signal to a specific process in a thread group. The `pthread_kill()` function of every POSIX-compliant *pthread* library invokes either of them to send a signal to a specific lightweight process.

The `tkill()` system call expects two parameters: the PID pid of the process to be signaled and the signal number sig. The `sys_tkill()` service routine fills a `siginfo` table, gets the process descriptor address, makes some permission checks (such as those in step 2 in the section "The group_send_sig_info() Function"), and invokes `specific_send_sig_info()` to send the signal.

The `tgkill()` system call differs from `tkill()` because it has a third parameter: the thread group ID (tgid) of the thread group that includes the process to be signaled. The `sys_tgkill()` service routine performs exactly the same operations as `sys_tkill()`, but also checks that the process being signaled actually belongs to the thread group tgid. This additional check solves a race condition that occurs when a signal is sent to a process that is being killed: if another multithreaded application is creating lightweight processes fast enough, the signal could be delivered to the wrong process. The `tgkill()` system call solves the problem, because the thread group ID is never changed during the life span of a multithreaded application.

Changing a Signal Action

The `sigaction(sig,act,oact)` system call allows users to specify an action for a signal; of course, if no signal action is defined, the kernel executes the default action associated with the delivered signal.

The corresponding `sys_sigaction()` service routine acts on two parameters: the sig signal number and the act table of type `old_sigaction` that specifies the new action.

A third oact optional output parameter may be used to get the previous action associated with the signal. (The old_sigaction data structure contains the same fields as the sigaction structure described in the earlier section "Data Structures Associated with Signals," but in a different order.)

The function checks first whether the act address is valid. Then it fills the sa_handler, sa_flags, and sa_mask fields of a new_ka local variable of type k_sigaction with the corresponding fields of *act:

```
__get_user(new_ka.sa.sa_handler, &act->sa_handler);
__get_user(new_ka.sa.sa_flags, &act->sa_flags);
__get_user(mask, &act->sa_mask);
siginitset(&new_ka.sa.sa_mask, mask);
```

The function invokes do_sigaction() to copy the new new_ka table into the entry at the sig-1 position of current->sig->action (the number of the signal is one higher than the position in the array because there is no zero signal):

```
k = &current->sig->action[sig-1];
if (act) {
    *k = *act;
    sigdelsetmask(&k->sa.sa_mask, sigmask(SIGKILL) | sigmask(SIGSTOP));
    if (k->sa.sa_handler == SIG_IGN || (k->sa.sa_handler == SIG_DFL &&
     (sig==SIGCONT || sig==SIGCHLD || sig==SIGWINCH || sig==SIGURG))) {
        rm_from_queue(sigmask(sig), &current->signal->shared_pending);
        t = current;
        do {
            rm_from_queue(sigmask(sig), &current->pending);
            recalc_sigpending_tsk(t);
            t = next_thread(t);
        } while (t != current);
    }
}
```

The POSIX standard requires that setting a signal action to either SIG_IGN or SIG_DFL when the default action is "ignore" causes every pending signal of the same type to be discarded. Moreover, notice that no matter what the requested masked signals are for the signal handler, SIGKILL and SIGSTOP are never masked.

The sigaction() system call also allows the user to initialize the sa_flags field in the sigaction table. We listed the values allowed for this field and the related meanings in Table 11-6 (earlier in this chapter).

Older System V Unix variants offered the signal() system call, which is still widely used by programmers. Recent C libraries implement signal() by means of rt_sigaction(). However, Linux still supports older C libraries and offers the sys_signal() service routine:

```
new_sa.sa.sa_handler = handler;
new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
ret = do_sigaction(sig, &new_sa, &old_sa);
return ret ? ret : (unsigned long)old_sa.sa.sa_handler;
```

Examining the Pending Blocked Signals

The sigpending( ) system call allows a process to examine the set of pending blocked signals—i.e., those that have been raised while blocked. The corresponding sys_ sigpending( ) service routine acts on a single parameter, set, namely, the address of a user variable where the array of bits must be copied:

```
sigorsets(&pending, &current->pending.signal,
                    &current->signal->shared_pending.signal);
sigandsets(&pending, &current->blocked, &pending);
copy_to_user(set, &pending, 4);
```

Modifying the Set of Blocked Signals

The sigprocmask( ) system call allows processes to modify the set of blocked signals; it applies only to regular (non-real-time) signals. The corresponding sys_ sigprocmask( ) service routine acts on three parameters:

oset

    Pointer in the process address space to a bit array where the previous bit mask must be stored.

set

    Pointer in the process address space to the bit array containing the new bit mask.

how

    Flag that may have one of the following values:

    SIG_BLOCK

        The *set bit mask array specifies the signals that must be added to the bit mask array of blocked signals.

    SIG_UNBLOCK

        The *set bit mask array specifies the signals that must be removed from the bit mask array of blocked signals.

    SIG_SETMASK

        The *set bit mask array specifies the new bit mask array of blocked signals.

The function invokes copy_from_user( ) to copy the value pointed to by the set parameter into the new_set local variable and copies the bit mask array of standard blocked signals of current into the old_set local variable. It then acts as the how flag specifies on these two variables:

```
if (copy_from_user(&new_set, set, sizeof(*set)))
    return -EFAULT;
new_set &= ~(sigmask(SIGKILL)|sigmask(SIGSTOP));
old_set = current->blocked.sig[0];
if (how == SIG_BLOCK)
    sigaddsetmask(&current->blocked, new_set);
else if (how == SIG_UNBLOCK)
    sigdelsetmask(&current->blocked, new_set);
```

```
    else if (how == SIG_SETMASK)
        current->blocked.sig[0] = new_set;
    else
        return -EINVAL;
recalc_sigpending(current);
if (oset && copy_to_user(oset, &old_set, sizeof(*oset)))
        return -EFAULT;
return 0;
```

Suspending the Process

The sigsuspend( ) system call puts the process in the TASK_INTERRUPTIBLE state, after having blocked the standard signals specified by a bit mask array to which the mask parameter points. The process will wake up only when a nonignored, nonblocked signal is sent to it.

The corresponding sys_sigsuspend( ) service routine executes these statements:

```
mask &= ~(sigmask(SIGKILL) | sigmask(SIGSTOP));
saveset = current->blocked;
siginitset(&current->blocked, mask);
recalc_sigpending(current);
regs->eax = -EINTR;
while (1) {
    current->state = TASK_INTERRUPTIBLE;
    schedule( );
    if (do_signal(regs, &saveset))
        return -EINTR;
}
```

The schedule( ) function selects another process to run. When the process that issued the sigsuspend( ) system call is executed again, sys_sigsuspend( ) invokes the do_signal( ) function to deliver the signal that has awakened the process. If that function returns the value 1, the signal is not ignored. Therefore the system call terminates by returning the error code -EINTR.

The sigsuspend( ) system call may appear redundant, because the combined execution of sigprocmask( ) and sleep( ) apparently yields the same result. But this is not true: because processes can be interleaved at any time, one must be conscious that invoking a system call to perform action A followed by another system call to perform action B is not equivalent to invoking a single system call that performs action A and then action B.

In this particular case, sigprocmask( ) might unblock a signal that is delivered before invoking sleep( ). If this happens, the process might remain in a TASK_INTERRUPTIBLE state forever, waiting for the signal that was already delivered. On the other hand, the sigsuspend( ) system call does not allow signals to be sent after unblocking and before the schedule( ) invocation, because other processes cannot grab the CPU during that time interval.

System Calls for Real-Time Signals

Because the system calls previously examined apply only to standard signals, additional system calls must be introduced to allow User Mode processes to handle real-time signals.

Several system calls for real-time signals (`rt_sigaction()`, `rt_sigpending()`, `rt_sigprocmask()`, and `rt_sigsuspend()`) are similar to those described earlier and won't be discussed further. For the same reason, we won't discuss two other system calls that deal with queues of real-time signals:

`rt_sigqueueinfo()`
> Sends a real-time signal so that it is added to the shared pending signal queue of the destination process. Usually invoked through the `sigqueue()` standard library function.

`rt_sigtimedwait()`
> Dequeues a blocked pending signal without delivering it and returns the signal number to the caller; if no blocked signal is pending, suspends the current process for a fixed amount of time. Usually invoked through the `sigwaitinfo()` and `sigtimedwait()` standard library functions.

CHAPTER 12

The Virtual Filesystem

One of Linux's keys to success is its ability to coexist comfortably with other systems. You can transparently mount disks or partitions that host file formats used by Windows, other Unix systems, or even systems with tiny market shares like the Amiga. Linux manages to support multiple filesystem types in the same way other Unix variants do, through a concept called the Virtual Filesystem.

The idea behind the Virtual Filesystem is to put a wide range of information in the kernel to represent many different types of filesystems; there is a field or function to support each operation provided by all real filesystems supported by Linux. For each read, write, or other function called, the kernel substitutes the actual function that supports a native Linux filesystem, the NTFS filesystem, or whatever other filesystem the file is on.

This chapter discusses the aims, structure, and implementation of Linux's Virtual Filesystem. It focuses on three of the five standard Unix file types—namely, regular files, directories, and symbolic links. Device files are covered in Chapter 13, while pipes are discussed in Chapter 19. To show how a real filesystem works, Chapter 18 covers the Second Extended Filesystem that appears on nearly all Linux systems.

The Role of the Virtual Filesystem (VFS)

The *Virtual Filesystem* (also known as Virtual Filesystem Switch or *VFS*) is a kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.

For instance, let's assume that a user issues the shell command:

```
$ cp /floppy/TEST /tmp/test
```

where */floppy* is the mount point of an MS-DOS diskette and */tmp* is a normal Second Extended Filesystem (Ext2) directory. The VFS is an abstraction layer between the application program and the filesystem implementations (see Figure 12-1(a)). Therefore, the *cp* program is not required to know the filesystem types of */floppy/TEST* and */tmp/test*. Instead, *cp* interacts with the VFS by means of generic system

calls known to anyone who has done Unix programming (see the section "File-Handling System Calls" in Chapter 1); the code executed by *cp* is shown in Figure 12-1(b).


```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
        O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```
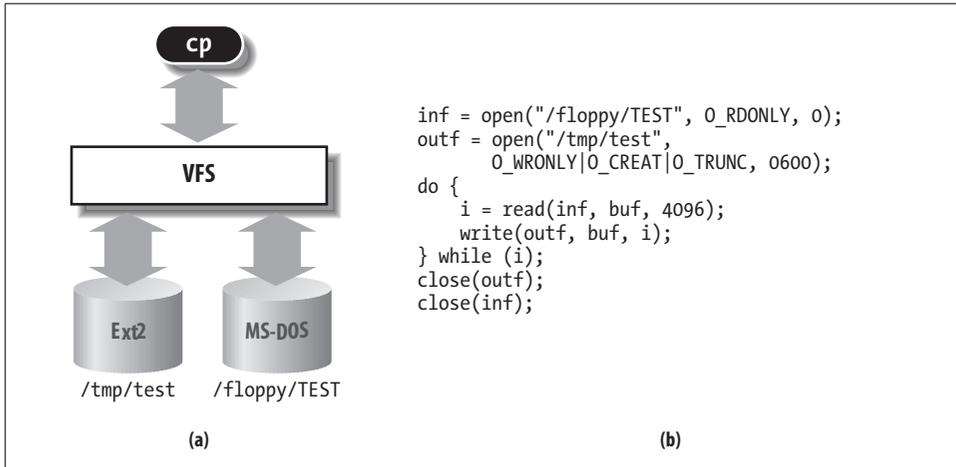
Figure 12-1. VFS role in a simple file copy operation

Filesystems supported by the VFS may be grouped into three main classes:

Disk-based filesystems

These manage memory space available in a local disk or in some other device that emulates a disk (such as a USB flash drive). Some of the well-known disk-based filesystems supported by the VFS are:

- Filesystems for Linux such as the widely used Second Extended Filesystem (Ext2), the recent Third Extended Filesystem (Ext3), and the Reiser Filesystems (ReiserFS)*

- Filesystems for Unix variants such as sysv filesystem (System V, Coherent, Xenix), UFS (BSD, Solaris, NEXTSTEP), MINIX filesystem, and VERITAS VxFS (SCO UnixWare)

- Microsoft filesystems such as MS-DOS, VFAT (Windows 95 and later releases), and NTFS (Windows NT 4 and later releases)

- ISO9660 CD-ROM filesystem (formerly High Sierra Filesystem) and Universal Disk Format (UDF) DVD filesystem

- Other proprietary filesystems such as IBM's OS/2 (HPFS), Apple's Macintosh (HFS), Amiga's Fast Filesystem (AFFS), and Acorn Disk Filing System (ADFS)

---

* Although these filesystems owe their birth to Linux, they have been ported to several other operating systems.

- Additional journaling filesystems originating in systems other than Linux such as IBM's JFS and SGI's XFS

Network filesystems

These allow easy access to files included in filesystems belonging to other networked computers. Some well-known network filesystems supported by the VFS are NFS, Coda, AFS (Andrew filesystem), CIFS (Common Internet File System, used in Microsoft Windows), and NCP (Novell's NetWare Core Protocol).

Special filesystems

These do not manage disk space, either locally or remotely. The */proc* filesystem is a typical example of a special filesystem (see the later section "Special Filesystems").

In this book, we describe in detail the Ext2 and Ext3 filesystems only (see Chapter 18); the other filesystems are not covered for lack of space.

As mentioned in the section "An Overview of the Unix Filesystem" in Chapter 1, Unix directories build a tree whose root is the / directory. The root directory is contained in the *root filesystem*, which in Linux, is usually of type Ext2 or Ext3. All other filesystems can be "mounted" on subdirectories of the root filesystem.[*]

A disk-based filesystem is usually stored in a hardware block device such as a hard disk, a floppy, or a CD-ROM. A useful feature of Linux's VFS allows it to handle *virtual block devices* such as */dev/loop0*, which may be used to mount filesystems stored in regular files. As a possible application, a user may protect her own private filesystem by storing an encrypted version of it in a regular file.

The first Virtual Filesystem was included in Sun Microsystems's SunOS in 1986. Since then, most Unix filesystems include a VFS. Linux's VFS, however, supports the widest range of filesystems.

The Common File Model

The key idea behind the VFS consists of introducing a *common file model* capable of representing all supported filesystems. This model strictly mirrors the file model provided by the traditional Unix filesystem. This is not surprising, because Linux wants to run its native filesystem with minimum overhead. However, each specific filesystem implementation must translate its physical organization into the VFS's common file model.

---

[*] When a filesystem is mounted on a directory, the contents of the directory in the parent filesystem are no longer accessible, because every pathname, including the mount point, will refer to the mounted filesystem. However, the original directory's content shows up again when the filesystem is unmounted. This somewhat surprising feature of Unix filesystems is used by system administrators to hide files; they simply mount a filesystem on the directory containing the files to be hidden.

For instance, in the common file model, each directory is regarded as a file, which contains a list of files and other directories. However, several non-Unix disk-based filesystems use a File Allocation Table (FAT), which stores the position of each file in the directory tree. In these filesystems, directories are not files. To stick to the VFS's common file model, the Linux implementations of such FAT-based filesystems must be able to construct on the fly, when needed, the files corresponding to the directories. Such files exist only as objects in kernel memory.

More essentially, the Linux kernel cannot hardcode a particular function to handle an operation such as `read( )` or `ioctl( )`. Instead, it must use a pointer for each operation; the pointer is made to point to the proper function for the particular filesystem being accessed.

Let's illustrate this concept by showing how the `read( )` shown in Figure 12-1 would be translated by the kernel into a call specific to the MS-DOS filesystem. The application's call to `read( )` makes the kernel invoke the corresponding `sys_read( )` service routine, like every other system call. The file is represented by a `file` data structure in kernel memory, as we'll see later in this chapter. This data structure contains a field called `f_op` that contains pointers to functions specific to MS-DOS files, including a function that reads a file. `sys_read( )` finds the pointer to this function and invokes it. Thus, the application's `read( )` is turned into the rather indirect call:

```
file->f_op->read(...);
```

Similarly, the `write( )` operation triggers the execution of a proper Ext2 write function associated with the output file. In short, the kernel is responsible for assigning the right set of pointers to the `file` variable associated with each open file, and then for invoking the call specific to each filesystem that the `f_op` field points to.

One can think of the common file model as object-oriented, where an *object* is a software construct that defines both a data structure and the methods that operate on it. For reasons of efficiency, Linux is not coded in an object-oriented language such as C++. Objects are therefore implemented as plain C data structures with some fields pointing to functions that correspond to the object's methods.

The common file model consists of the following object types:

The superblock object
    Stores information concerning a mounted filesystem. For disk-based filesystems, this object usually corresponds to a *filesystem control block* stored on disk.

The inode object
    Stores general information about a specific file. For disk-based filesystems, this object usually corresponds to a *file control block* stored on disk. Each inode object is associated with an *inode number*, which uniquely identifies the file within the filesystem.

The file object
> Stores information about the interaction between an open file and a process. This information exists only in kernel memory during the period when a process has the file open.

The dentry object
> Stores information about the linking of a directory entry (that is, a particular name of the file) with the corresponding file. Each disk-based filesystem stores this information in its own particular way on disk.

Figure 12-2 illustrates with a simple example how processes interact with files. Three different processes have opened the same file, two of them using the same hard link. In this case, each of the three processes uses its own file object, while only two dentry objects are required—one for each hard link. Both dentry objects refer to the same inode object, which identifies the superblock object and, together with the latter, the common disk file.
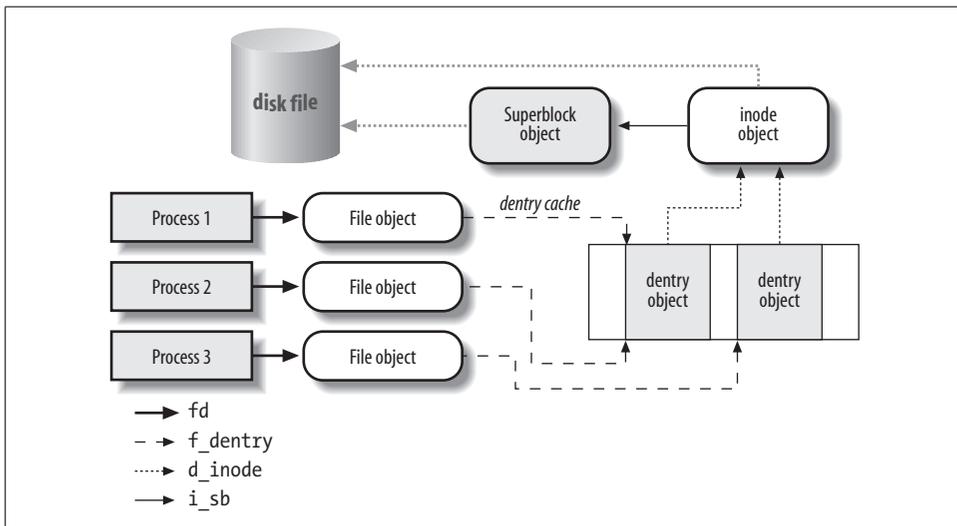
Figure 12-2. Interaction between processes and VFS objects

Besides providing a common interface to all filesystem implementations, the VFS has another important role related to system performance. The most recently used dentry objects are contained in a disk cache named the *dentry cache*, which speeds up the translation from a file pathname to the inode of the last pathname component.

Generally speaking, a *disk cache* is a software mechanism that allows the kernel to keep in RAM some information that is normally stored on a disk, so that further accesses to that data can be quickly satisfied without a slow access to the disk itself.

Notice how a disk cache differs from a hardware cache or a memory cache, neither of which has anything to do with disks or other devices. A hardware cache is a fast

static RAM that speeds up requests directed to the slower dynamic RAM (see the section "Hardware Cache" in Chapter 2). A memory cache is a software mechanism introduced to bypass the Kernel Memory Allocator (see the section "The Slab Allocator" in Chapter 8).

Beside the dentry cache and the inode cache, Linux uses other disk caches. The most important one, called the page cache, is described in detail in Chapter 15.

System Calls Handled by the VFS

Table 12-1 illustrates the VFS system calls that refer to filesystems, regular files, directories, and symbolic links. A few other system calls handled by the VFS, such as ioperm( ), ioctl( ), pipe( ), and mknod( ), refer to device files and pipes. These are discussed in later chapters. A last group of system calls handled by the VFS, such as socket( ), connect( ), and bind( ), refer to sockets and are used to implement networking. Some of the kernel service routines that correspond to the system calls listed in Table 12-1 are discussed either in this chapter or in Chapter 18.

Table 12-1. Some system calls handled by the VFS

System call name	Description
mount( ) umount( ) umount2( )	Mount/unmount filesystems
sysfs( )	Get filesystem information
statfs( ) fstatfs( ) statfs64( ) fstatfs64( ) ustat( )	Get filesystem statistics
chroot( ) pivot_root( )	Change root directory
chdir( ) fchdir( ) getcwd( )	Manipulate current directory
mkdir( ) rmdir( )	Create and destroy directories
getdents( ) getdents64( ) readdir( ) link( ) unlink( ) rename( ) lookup_dcookie( )	Manipulate directory entries
readlink( ) symlink( )	Manipulate soft links
chown( ) fchown( ) lchown( ) chown16( ) fchown16( ) lchown16( )	Modify file owner
chmod( ) fchmod( ) utime( )	Modify file attributes
stat( ) fstat( ) lstat( ) access( ) oldstat( ) oldfstat( ) oldlstat( ) stat64( ) lstat64( ) fstat64( )	Read file status
open( ) close( ) creat( ) umask( )	Open, close, and create files
dup( ) dup2( ) fcntl( ) fcntl64( )	Manipulate file descriptors
select( ) poll( )	Wait for events on a set of file descriptors
truncate( ) ftruncate( ) truncate64( ) ftruncate64( )	Change file size
lseek( ) _llseek( )	Change file pointer
read( ) write( ) readv( ) writev( ) sendfile( ) sendfile64( ) readahead( )	Carry out file I/O operations

Table 12-1. Some system calls handled by the VFS (continued)

System call name	Description
`io_setup() io_submit() io_getevents() io_cancel() io_destroy()`	Asynchronous I/O (allows multiple outstanding read and write requests)
`pread64() pwrite64()`	Seek file and access it
`mmap() mmap2() munmap() madvise() mincore() remap_file_pages()`	Handle file memory mapping
`fdatasync() fsync() sync() msync()`	Synchronize file data
`flock()`	Manipulate file lock
`setxattr() lsetxattr() fsetxattr() getxattr() lgetxattr() fgetxattr() listxattr() llistxattr() flistxattr() removexattr() lremovexattr() fremovexattr()`	Manipulate file extended attributes

We said earlier that the VFS is a layer between application programs and specific file-systems. However, in some cases, a file operation can be performed by the VFS itself, without invoking a lower-level procedure. For instance, when a process closes an open file, the file on disk doesn't usually need to be touched, and hence the VFS simply releases the corresponding file object. Similarly, when the lseek() system call modifies a file pointer, which is an attribute related to the interaction between an opened file and a process, the VFS needs to modify only the corresponding file object without accessing the file on disk, and therefore it does not have to invoke a specific filesystem procedure. In some sense, the VFS could be considered a "generic" filesystem that relies, when necessary, on specific ones.

VFS Data Structures

Each VFS object is stored in a suitable data structure, which includes both the object attributes and a pointer to a table of object methods. The kernel may dynamically modify the methods of the object and, hence, it may install specialized behavior for the object. The following sections explain the VFS objects and their interrelationships in detail.

Superblock Objects

A superblock object consists of a `super_block` structure whose fields are described in Table 12-2.

Table 12-2. The fields of the superblock object

Type	Field	Description
`struct list_head`	`s_list`	Pointers for superblock list
`dev_t`	`s_dev`	Device identifier

Table 12-2. The fields of the superblock object (continued)

Type	Field	Description
unsigned long	s_blocksize	Block size in bytes
unsigned long	s_old_blocksize	Block size in bytes as reported by the underlying block device driver
unsigned char	s_blocksize_bits	Block size in number of bits
unsigned char	s_dirt	Modified (dirty) flag
unsigned long long	s_maxbytes	Maximum size of the files
struct file_system_type *	s_type	Filesystem type
struct super_operations *	s_op	Superblock methods
struct dquot_operations *	dq_op	Disk quota handling methods
struct quotactl_ops *	s_qcop	Disk quota administration methods
struct export_operations *	s_export_op	Export operations used by network filesystems
unsigned long	s_flags	Mount flags
unsigned long	s_magic	Filesystem magic number
struct dentry *	s_root	Dentry object of the filesystem's root directory
struct rw_semaphore	s_umount	Semaphore used for unmounting
struct semaphore	s_lock	Superblock semaphore
int	s_count	Reference counter
int	s_syncing	Flag indicating that inodes of the superblock are being synchronized
int	s_need_sync_fs	Flag used when synchronizing the superblock's mounted filesystem
atomic_t	s_active	Secondary reference counter
void *	s_security	Pointer to superblock security structure
struct xattr_handler **	s_xattr	Pointer to superblock extended attribute structure
struct list_head	s_inodes	List of all inodes
struct list_head	s_dirty	List of modified inodes
struct list_head	s_io	List of inodes waiting to be written to disk
struct hlist_head	s_anon	List of anonymous dentries for handling remote network filesystems
struct list_head	s_files	List of file objects
struct block_device *	s_bdev	Pointer to the block device driver descriptor
struct list_head	s_instances	Pointers for a list of superblock objects of a given filesystem type (see the later section "Filesystem Type Registration")

Table 12-2. The fields of the superblock object (continued)

Type	Field	Description
struct quota_info	s_dquot	Descriptor for disk quota
int	s_frozen	Flag used when freezing the filesystem (forcing it to a consistent state)
wait_queue_head_t	s_wait_unfrozen	Wait queue where processes sleep until the filesystem is unfrozen
char[]	s_id	Name of the block device containing the superblock
void *	s_fs_info	Pointer to superblock information of a specific filesystem
struct semaphore	s_vfs_rename_sem	Semaphore used by VFS when renaming files across directories
u32	s_time_gran	Timestamp's granularity (in nanoseconds)

All superblock objects are linked in a circular doubly linked list. The first element of this list is represented by the super_blocks variable, while the s_list field of the superblock object stores the pointers to the adjacent elements in the list. The sb_lock spin lock protects the list against concurrent accesses in multiprocessor systems.

The s_fs_info field points to superblock information that belongs to a specific filesystem; for instance, as we'll see later in Chapter 18, if the superblock object refers to an Ext2 filesystem, the field points to an ext2_sb_info structure, which includes the disk allocation bit masks and other data of no concern to the VFS common file model.

In general, data pointed to by the s_fs_info field is information from the disk duplicated in memory for reasons of efficiency. Each disk-based filesystem needs to access and update its allocation bitmaps in order to allocate or release disk blocks. The VFS allows these filesystems to act directly on the s_fs_info field of the superblock in memory without accessing the disk.

This approach leads to a new problem, however: the VFS superblock might end up no longer synchronized with the corresponding superblock on disk. It is thus necessary to introduce an s_dirt flag, which specifies whether the superblock is dirty—that is, whether the data on the disk must be updated. The lack of synchronization leads to the familiar problem of a corrupted filesystem when a site's power goes down without giving the user the chance to shut down a system cleanly. As we'll see in the section "Writing Dirty Pages to Disk" in Chapter 15, Linux minimizes this problem by periodically copying all dirty superblocks to disk.

The methods associated with a superblock are called *superblock operations*. They are described by the super_operations structure whose address is included in the s_op field.

Each specific filesystem can define its own superblock operations. When the VFS needs to invoke one of them, say read_inode( ), it executes the following:

```
sb->s_op->read_inode(inode);
```

where `sb` stores the address of the superblock object involved. The `read_inode` field of the `super_operations` table contains the address of the suitable function, which is therefore directly invoked.

Let's briefly describe the superblock operations, which implement higher-level operations like deleting files or mounting disks. They are listed in the order they appear in the `super_operations` table:

`alloc_inode(sb)`
 Allocates space for an inode object, including the space required for filesystem-specific data.

`destroy_inode(inode)`
 Destroys an inode object, including the filesystem-specific data.

`read_inode(inode)`
 Fills the fields of the inode object passed as the parameter with the data on disk; the `i_ino` field of the inode object identifies the specific filesystem inode on the disk to be read.

`dirty_inode(inode)`
 Invoked when the inode is marked as modified (dirty). Used by filesystems such as ReiserFS and Ext3 to update the filesystem journal on disk.

`write_inode(inode, flag)`
 Updates a filesystem inode with the contents of the inode object passed as the parameter; the `i_ino` field of the inode object identifies the filesystem inode on disk that is concerned. The `flag` parameter indicates whether the I/O operation should be synchronous.

`put_inode(inode)`
 Invoked when the inode is released—its reference counter is decreased—to perform filesystem-specific operations.

`drop_inode(inode)`
 Invoked when the inode is about to be destroyed—that is, when the last user releases the inode; filesystems that implement this method usually make use of `generic_drop_inode()`. This function removes every reference to the inode from the VFS data structures and, if the inode no longer appears in any directory, invokes the `delete_inode` superblock method to delete the inode from the filesystem.

`delete_inode(inode)`
 Invoked when the inode must be destroyed. Deletes the VFS inode in memory and the file data and metadata on disk.

`put_super(super)`
 Releases the superblock object passed as the parameter (because the corresponding filesystem is unmounted).

`write_super(super)`
 Updates a filesystem superblock with the contents of the object indicated.

`sync_fs(sb, wait)`
    Invoked when flushing the filesystem to update filesystem-specific data structures on disk (used by journaling filesystems).

`write_super_lockfs(super)`
    Blocks changes to the filesystem and updates the superblock with the contents of the object indicated. This method is invoked when the filesystem is frozen, for instance by the Logical Volume Manager (LVM) driver.

`unlockfs(super)`
    Undoes the block of filesystem updates achieved by the `write_super_lockfs` superblock method.

`statfs(super, buf)`
    Returns statistics on a filesystem by filling the `buf` buffer.

`remount_fs(super, flags, data)`
    Remounts the filesystem with new options (invoked when a mount option must be changed).

`clear_inode(inode)`
    Invoked when a disk inode is being destroyed to perform filesystem-specific operations.

`umount_begin(super)`
    Aborts a mount operation because the corresponding unmount operation has been started (used only by network filesystems).

`show_options(seq_file, vfsmount)`
    Used to display the filesystem-specific options

`quota_read(super, type, data, size, offset)`
    Used by the quota system to read data from the file that specifies the limits for this filesystem.[*]

`quota_write(super, type, data, size, offset)`
    Used by the quota system to write data into the file that specifies the limits for this filesystem.

The preceding methods are available to all possible filesystem types. However, only a subset of them applies to each specific filesystem; the fields corresponding to unimplemented methods are set to `NULL`. Notice that no `get_super` method to read a superblock is defined—how could the kernel invoke a method of an object yet to be read from disk? We'll find an equivalent `get_sb` method in another object describing the filesystem type (see the later section "Filesystem Type Registration").

---

[*] The *quota system* defines for each user and/or group limits on the amount of space that can be used on a given filesystem (see the `quotactl()` system call.)

Inode Objects

All information needed by the filesystem to handle a file is included in a data structure called an inode. A filename is a casually assigned label that can be changed, but the inode is unique to the file and remains the same as long as the file exists. An inode object in memory consists of an inode structure whose fields are described in Table 12-3.

Table 12-3. The fields of the inode object

Type	Field	Description
struct hlist_node	i_hash	Pointers for the hash list
struct list_head	i_list	Pointers for the list that describes the inode's current state
struct list_head	i_sb_list	Pointers for the list of inodes of the superblock
struct list_head	i_dentry	The head of the list of dentry objects referencing this inode
unsigned long	i_ino	inode number
atomic_t	i_count	Usage counter
umode_t	i_mode	File type and access rights
unsigned int	i_nlink	Number of hard links
uid_t	i_uid	Owner identifier
gid_t	i_gid	Group identifier
dev_t	i_rdev	Real device identifier
loff_t	i_size	File length in bytes
struct timespec	i_atime	Time of last file access
struct timespec	i_mtime	Time of last file write
struct timespec	i_ctime	Time of last inode change
unsigned int	i_blkbits	Block size in number of bits
unsigned long	i_blksize	Block size in bytes
unsigned long	i_version	Version number, automatically increased after each use
unsigned long	i_blocks	Number of blocks of the file
unsigned short	i_bytes	Number of bytes in the last block of the file
unsigned char	i_sock	Nonzero if file is a socket
spinlock_t	i_lock	Spin lock protecting some fields of the inode
struct semaphore	i_sem	inode semaphore
struct rw_semaphore	i_alloc_sem	Read/write semaphore protecting against race conditions in direct I/O file operations
struct inode_operations *	i_op	inode operations
struct file_operations *	i_fop	Default file operations
struct super_block *	i_sb	Pointer to superblock object

Table 12-3. The fields of the inode object (continued)

Type	Field	Description
struct file_lock *	i_flock	Pointer to file lock list
struct address_space *	i_mapping	Pointer to an `address_space` object (see Chapter 15)
struct address_space	i_data	`address_space` object of the file
struct dquot * []	i_dquot	inode disk quotas
struct list_head	i_devices	Pointers for a list of inodes relative to a specific character or block device (see Chapter 13)
struct pipe_inode_info *	i_pipe	Used if the file is a pipe (see Chapter 19)
struct block_device *	i_bdev	Pointer to the block device driver
struct cdev *	i_cdev	Pointer to the character device driver
int	i_cindex	Index of the device file within a group of minor numbers
__u32	i_generation	inode version number (used by some filesystems)
unsigned long	i_dnotify_mask	Bit mask of directory notify events
struct dnotify_struct *	i_dnotify	Used for directory notifications
unsigned long	i_state	inode state flags
unsigned long	dirtied_when	Dirtying time (in ticks) of the inode
unsigned int	i_flags	Filesystem mount flags
atomic_t	i_writecount	Usage counter for writing processes
void *	i_security	Pointer to inode's security structure
void *	u.generic_ip	Pointer to private data
seqcount_t	i_size_seqcount	Sequence counter used in SMP systems to get consistent values for `i_size`

Each inode object duplicates some of the data included in the disk inode—for instance, the number of blocks allocated to the file. When the value of the i_state field is equal to I_DIRTY_SYNC, I_DIRTY_DATASYNC, or I_DIRTY_PAGES, the inode is dirty—that is, the corresponding disk inode must be updated. The I_DIRTY macro can be used to check the value of these three flags at once (see later for details). Other values of the i_state field are I_LOCK (the inode object is involved in an I/O transfer), I_FREEING (the inode object is being freed), I_CLEAR (the inode object contents are no longer meaningful), and I_NEW (the inode object has been allocated but not yet filled with data read from the disk inode).

Each inode object always appears in one of the following circular doubly linked lists (in all cases, the pointers to the adjacent elements are stored in the i_list field):

- The list of valid unused inodes, typically those mirroring valid disk inodes and not currently used by any process. These inodes are not dirty and their i_count field is set to 0. The first and last elements of this list are referenced by the next and prev fields, respectively, of the inode_unused variable. This list acts as a disk cache.

- The list of in-use inodes, that is, those mirroring valid disk inodes and used by some process. These inodes are not dirty and their i_count field is positive. The first and last elements are referenced by the inode_in_use variable.

- The list of dirty inodes. The first and last elements are referenced by the s_dirty field of the corresponding superblock object.

Each of the lists just mentioned links the i_list fields of the proper inode objects.

Moreover, each inode object is also included in a per-filesystem doubly linked circular list headed at the s_inodes field of the superblock object; the i_sb_list field of the inode object stores the pointers for the adjacent elements in this list.

Finally, the inode objects are also included in a hash table named inode_hashtable. The hash table speeds up the search of the inode object when the kernel knows both the inode number and the address of the superblock object corresponding to the file-system that includes the file. Because hashing may induce collisions, the inode object includes an i_hash field that contains a backward and a forward pointer to other inodes that hash to the same position; this field creates a doubly linked list of those inodes.

The methods associated with an inode object are also called *inode operations*. They are described by an inode_operations structure, whose address is included in the i_op field. Here are the inode operations in the order they appear in the inode_operations table:

create(dir, dentry, mode, nameidata)
> Creates a new disk inode for a regular file associated with a dentry object in some directory.

lookup(dir, dentry, nameidata)
> Searches a directory for an inode corresponding to the filename included in a dentry object.

link(old_dentry, dir, new_dentry)
> Creates a new hard link that refers to the file specified by old_dentry in the directory dir; the new hard link has the name specified by new_dentry.

unlink(dir, dentry)
> Removes the hard link of the file specified by a dentry object from a directory.

symlink(dir, dentry, symname)
> Creates a new inode for a symbolic link associated with a dentry object in some directory.

mkdir(dir, dentry, mode)
> Creates a new inode for a directory associated with a dentry object in some directory.

rmdir(dir, dentry)
> Removes from a directory the subdirectory whose name is included in a dentry object.

`mknod(dir, dentry, mode, rdev)`

Creates a new disk inode for a special file associated with a dentry object in some directory. The `mode` and `rdev` parameters specify, respectively, the file type and the device's major and minor numbers.

`rename(old_dir, old_dentry, new_dir, new_dentry)`

Moves the file identified by `old_entry` from the `old_dir` directory to the `new_dir` one. The new filename is included in the dentry object that `new_dentry` points to.

`readlink(dentry, buffer, buflen)`

Copies into a User Mode memory area specified by `buffer` the file pathname corresponding to the symbolic link specified by the dentry.

`follow_link(inode, nameidata)`

Translates a symbolic link specified by an inode object; if the symbolic link is a relative pathname, the lookup operation starts from the directory specified in the second parameter.

`put_link(dentry, nameidata)`

Releases all temporary data structures allocated by the `follow_link` method to translate a symbolic link.

`truncate(inode)`

Modifies the size of the file associated with an inode. Before invoking this method, it is necessary to set the `i_size` field of the inode object to the required new size.

`permission(inode, mask, nameidata)`

Checks whether the specified access mode is allowed for the file associated with `inode`.

`setattr(dentry, iattr)`

Notifies a "change event" after touching the inode attributes.

`getattr(mnt, dentry, kstat)`

Used by some filesystems to read inode attributes.

`setxattr(dentry, name, value, size, flags)`

Sets an "extended attribute" of an inode (extended attributes are stored on disk blocks outside of any inode).

`getxattr(dentry, name, buffer, size)`

Gets an extended attribute of an inode.

`listxattr(dentry, buffer, size)`

Gets the whole list of extended attribute names.

`removexattr(dentry, name)`

Removes an extended attribute of an inode.

The methods just listed are available to all possible inodes and filesystem types. However, only a subset of them applies to a specific inode and filesystem; the fields corresponding to unimplemented methods are set to `NULL`.

File Objects

A file object describes how a process interacts with a file it has opened. The object is created when the file is opened and consists of a `file` structure, whose fields are described in Table 12-4. Notice that file objects have no corresponding image on disk, and hence no "dirty" field is included in the `file` structure to specify that the file object has been modified.

Table 12-4. The fields of the file object

Type	Field	Description
struct list_head	f_list	Pointers for generic file object list
struct dentry *	f_dentry	dentry object associated with the file
struct vfsmount *	f_vfsmnt	Mounted filesystem containing the file
struct file_operations *	f_op	Pointer to file operation table
atomic_t	f_count	File object's reference counter
unsigned int	f_flags	Flags specified when opening the file
mode_t	f_mode	Process access mode
int	f_error	Error code for network write operation
loff_t	f_pos	Current file offset (file pointer)
struct fown_struct	f_owner	Data for I/O event notification via signals
unsigned int	f_uid	User's UID
unsigned int	f_gid	User group ID
struct file_ra_state	f_ra	File read-ahead state (see Chapter 16)
size_t	f_maxcount	Maximum number of bytes that can be read or written with a single operation (currently set to $2^{31}-1$)
unsigned long	f_version	Version number, automatically increased after each use
void *	f_security	Pointer to file object's security structure
void *	private_data	Pointer to data specific for a filesystem or a device driver
struct list_head	f_ep_links	Head of the list of event poll waiters for this file
spinlock_t	f_ep_lock	Spin lock protecting the f_ep_links list
struct address_space *	f_mapping	Pointer to file's address space object (see Chapter 15)

The main information stored in a file object is the *file pointer*—the current position in the file from which the next operation will take place. Because several processes may access the same file concurrently, the file pointer must be kept in the file object rather than the inode object.

File objects are allocated through a slab cache named *filp*, whose descriptor address is stored in the `filp_cachep` variable. Because there is a limit on the number of file objects that can be allocated, the `files_stat` variable specifies in the `max_files` field

the maximum number of allocatable file objects—i.e., the maximum number of files that can be accessed at the same time in the system.*

"In use" file objects are collected in several lists rooted at the superblocks of the owning filesystems. Each superblock object stores in the s_files field the head of a list of file objects; thus, file objects of files belonging to different filesystems are included in different lists. The pointers to the previous and next element in the list are stored in the f_list field of the file object. The files_lock spin lock protects the superblock s_files lists against concurrent accesses in multiprocessor systems.

The f_count field of the file object is a reference counter: it counts the number of processes that are using the file object (remember that lightweight processes created with the CLONE_FILES flag share the table that identifies the open files, thus they use the same file objects). The counter is also increased when the file object is used by the kernel itself—for instance, when the object is inserted in a list, or when a dup() system call has been issued.

When the VFS must open a file on behalf of a process, it invokes the get_empty_filp( ) function to allocate a new file object. The function invokes kmem_cache_alloc() to get a free file object from the *filp* cache, then it initializes the fields of the object as follows:

```
memset(f, 0, sizeof(*f));
INIT_LIST_HEAD(&f->f_ep_links);
spin_lock_init(&f->f_ep_lock);
atomic_set(&f->f_count, 1);
f->f_uid = current->fsuid;
f->f_gid = current->fsgid;
f->f_owner.lock = RW_LOCK_UNLOCKED;
INIT_LIST_HEAD(&f->f_list);
f->f_maxcount = INT_MAX;
```

As we explained earlier in the section "The Common File Model," each filesystem includes its own set of *file operations* that perform such activities as reading and writing a file. When the kernel loads an inode into memory from disk, it stores a pointer to these file operations in a file_operations structure whose address is contained in the i_fop field of the inode object. When a process opens the file, the VFS initializes the f_op field of the new file object with the address stored in the inode so that further calls to file operations can use these functions. If necessary, the VFS may later modify the set of file operations by storing a new value in f_op.

The following list describes the file operations in the order in which they appear in the file_operations table:

---

* The files_init() function, executed during kernel initialization, sets the max_files field to one-tenth of the available RAM in kilobytes, but the system administrator can tune this parameter by writing into the */proc/sys/fs/file-max* file. Moreover, the superuser can always get a file object, even if max_files file objects have already been allocated.

`llseek(file, offset, origin)`
> Updates the file pointer.

`read(file, buf, count, offset)`
> Reads count bytes from a file starting at position *offset; the value *offset (which usually corresponds to the file pointer) is then increased.

`aio_read(req, buf, len, pos)`
> Starts an asynchronous I/O operation to read `len` bytes into `buf` from file position pos (introduced to support the `io_submit()` system call).

`write(file, buf, count, offset)`
> Writes count bytes into a file starting at position *offset; the value *offset (which usually corresponds to the file pointer) is then increased.

`aio_write(req, buf, len, pos)`
> Starts an asynchronous I/O operation to write `len` bytes from `buf` to file position pos.

`readdir(dir, dirent, filldir)`
> Returns the next directory entry of a directory in `dirent`; the `filldir` parameter contains the address of an auxiliary function that extracts the fields in a directory entry.

`poll(file, poll_table)`
> Checks whether there is activity on a file and goes to sleep until something happens on it.

`ioctl(inode, file, cmd, arg)`
> Sends a command to an underlying hardware device. This method applies only to device files.

`unlocked_ioctl(file, cmd, arg)`
> Similar to the `ioctl` method, but it does not take the big kernel lock (see the section "The Big Kernel Lock" in Chapter 5). It is expected that all device drivers and all filesystems will implement this new method instead of the `ioctl` method.

`compat_ioctl(file, cmd, arg)`
> Method used to implement the `ioctl()` 32-bit system call by 64-bit kernels.

`mmap(file, vma)`
> Performs a memory mapping of the file into a process address space (see the section "Memory Mapping" in Chapter 16).

`open(inode, file)`
> Opens a file by creating a new file object and linking it to the corresponding inode object (see the section "The open( ) System Call" later in this chapter).

`flush(file)`
> Called when a reference to an open file is closed. The actual purpose of this method is filesystem-dependent.

`release(inode, file)`
> Releases the file object. Called when the last reference to an open file is closed—that is, when the `f_count` field of the file object becomes 0.

`fsync(file, dentry, flag)`
> Flushes the file by writing all cached data to disk.

`aio_fsync(req, flag)`
> Starts an asynchronous I/O flush operation.

`fasync(fd, file, on)`
> Enables or disables I/O event notification by means of signals.

`lock(file, cmd, file_lock)`
> Applies a lock to the file (see the section "File Locking" later in this chapter).

`readv(file, vector, count, offset)`
> Reads bytes from a file and puts the results in the buffers described by `vector`; the number of buffers is specified by `count`.

`writev(file, vector, count, offset)`
> Writes bytes into a file from the buffers described by `vector`; the number of buffers is specified by `count`.

`sendfile(in_file, offset, count, file_send_actor, out_file)`
> Transfers data from `in_file` to `out_file` (introduced to support the `sendfile()` system call).

`sendpage(file, page, offset, size, pointer, fill)`
> Transfers data from `file` to the page cache's page; this is a low-level method used by `sendfile()` and by the networking code for sockets.

`get_unmapped_area(file, addr, len, offset, flags)`
> Gets an unused address range to map the file.

`check_flags(flags)`
> Method invoked by the service routine of the `fcntl()` system call to perform additional checks when setting the status flags of a file (`F_SETFL` command). Currently used only by the NFS network filesystem.

`dir_notify(file, arg)`
> Method invoked by the service routine of the `fcntl()` system call when establishing a directory change notification (`F_NOTIFY` command). Currently used only by the Common Internet File System (CIFS) network filesystem.

`flock(file, flag, lock)`
> Used to customize the behavior of the `flock()` system call. No official Linux filesystem makes use of this method.

The methods just described are available to all possible file types. However, only a subset of them apply to a specific file type; the fields corresponding to unimplemented methods are set to `NULL`.

dentry Objects

We mentioned in the section "The Common File Model" that the VFS considers each directory a file that contains a list of files and other directories. We will discuss in Chapter 18 how directories are implemented on a specific filesystem. Once a directory entry is read into memory, however, it is transformed by the VFS into a dentry object based on the dentry structure, whose fields are described in Table 12-5. The kernel creates a dentry object for every component of a pathname that a process looks up; the dentry object associates the component to its corresponding inode. For example, when looking up the */tmp/test* pathname, the kernel creates a dentry object for the / root directory, a second dentry object for the *tmp* entry of the root directory, and a third dentry object for the *test* entry of the */tmp* directory.

Notice that dentry objects have no corresponding image on disk, and hence no field is included in the dentry structure to specify that the object has been modified. Dentry objects are stored in a slab allocator cache whose descriptor is dentry_cache; dentry objects are thus created and destroyed by invoking kmem_cache_alloc() and kmem_cache_free().

Table 12-5. The fields of the dentry object

Type	Field	Description
atomic_t	d_count	Dentry object usage counter
unsigned int	d_flags	Dentry cache flags
spinlock_t	d_lock	Spin lock protecting the dentry object
struct inode *	d_inode	Inode associated with filename
struct dentry *	d_parent	Dentry object of parent directory
struct qstr	d_name	Filename
struct list_head	d_lru	Pointers for the list of unused dentries
struct list_head	d_child	For directories, pointers for the list of directory dentries in the same parent directory
struct list_head	d_subdirs	For directories, head of the list of subdirectory dentries
struct list_head	d_alias	Pointers for the list of dentries associated with the same inode (alias)
unsigned long	d_time	Used by d_revalidate method
struct dentry_operations*	d_op	Dentry methods
struct super_block *	d_sb	Superblock object of the file
void *	d_fsdata	Filesystem-dependent data
struct rcu_head	d_rcu	The RCU descriptor used when reclaiming the dentry object (see the section "Read-Copy Update (RCU)" in Chapter 5)
struct dcookie_struct *	d_cookie	Pointer to structure used by kernel profilers
struct hlist_node	d_hash	Pointer for list in hash table entry

Table 12-5. The fields of the dentry object (continued)

Type	Field	Description
int	d_mounted	For directories, counter for the number of filesystems mounted on this dentry
unsigned char[]	d_iname	Space for short filename

Each dentry object may be in one of four states:

Free

> The dentry object contains no valid information and is not used by the VFS. The corresponding memory area is handled by the slab allocator.

Unused

> The dentry object is not currently used by the kernel. The d_count usage counter of the object is 0, but the d_inode field still points to the associated inode. The dentry object contains valid information, but its contents may be discarded if necessary in order to reclaim memory.

In use

> The dentry object is currently used by the kernel. The d_count usage counter is positive, and the d_inode field points to the associated inode object. The dentry object contains valid information and cannot be discarded.

Negative

> The inode associated with the dentry does not exist, either because the corresponding disk inode has been deleted or because the dentry object was created by resolving a pathname of a nonexistent file. The d_inode field of the dentry object is set to NULL, but the object still remains in the dentry cache, so that further lookup operations to the same file pathname can be quickly resolved. The term "negative" is somewhat misleading, because no negative value is involved.

The methods associated with a dentry object are called *dentry operations*; they are described by the dentry_operations structure, whose address is stored in the d_op field. Although some filesystems define their own dentry methods, the fields are usually NULL and the VFS replaces them with default functions. Here are the methods, in the order they appear in the dentry_operations table:

d_revalidate(dentry, nameidata)

> Determines whether the dentry object is still valid before using it for translating a file pathname. The default VFS function does nothing, although network filesystems may specify their own functions.

d_hash(dentry, name)

> Creates a hash value; this function is a filesystem-specific hash function for the dentry hash table. The dentry parameter identifies the directory containing the component. The name parameter points to a structure containing both the pathname component to be looked up and the value produced by the hash function.

`d_compare(dir, name1, name2)`

    Compares two filenames; `name1` should belong to the directory referenced by `dir`. The default VFS function is a normal string match. However, each filesystem can implement this method in its own way. For instance, MS-DOS does not distinguish capital from lowercase letters.

`d_delete(dentry)`

    Called when the last reference to a dentry object is deleted (`d_count` becomes 0). The default VFS function does nothing.

`d_release(dentry)`

    Called when a dentry object is going to be freed (released to the slab allocator). The default VFS function does nothing.

`d_iput(dentry, ino)`

    Called when a dentry object becomes "negative"—that is, it loses its inode. The default VFS function invokes `iput( )` to release the inode object.

The dentry Cache

Because reading a directory entry from disk and constructing the corresponding dentry object requires considerable time, it makes sense to keep in memory dentry objects that you've finished with but might need later. For instance, people often edit a file and then compile it, or edit and print it, or copy it and then edit the copy. In such cases, the same file needs to be repeatedly accessed.

To maximize efficiency in handling dentries, Linux uses a dentry cache, which consists of two kinds of data structures:

- A set of dentry objects in the in-use, unused, or negative state.
- A hash table to derive the dentry object associated with a given filename and a given directory quickly. As usual, if the required object is not included in the dentry cache, the search function returns a null value.

The dentry cache also acts as a controller for an *inode cache*. The inodes in kernel memory that are associated with unused dentries are not discarded, because the dentry cache is still using them. Thus, the inode objects are kept in RAM and can be quickly referenced by means of the corresponding dentries.

All the "unused" dentries are included in a doubly linked "Least Recently Used" list sorted by time of insertion. In other words, the dentry object that was last released is put in front of the list, so the least recently used dentry objects are always near the end of the list. When the dentry cache has to shrink, the kernel removes elements from the tail of this list so that the most recently used objects are preserved. The addresses of the first and last elements of the LRU list are stored in the `next` and `prev` fields of the `dentry_unused` variable of type `list_head`. The `d_lru` field of the dentry object contains pointers to the adjacent dentries in the list.

Each "in use" dentry object is inserted into a doubly linked list specified by the `i_dentry` field of the corresponding inode object (because each inode could be associated with several hard links, a list is required). The `d_alias` field of the dentry object stores the addresses of the adjacent elements in the list. Both fields are of type `struct list_head`.

An "in use" dentry object may become "negative" when the last hard link to the corresponding file is deleted. In this case, the dentry object is moved into the LRU list of unused dentries. Each time the kernel shrinks the dentry cache, negative dentries move toward the tail of the LRU list so that they are gradually freed (see the section "Reclaiming Pages of Shrinkable Disk Caches" in Chapter 17).

The hash table is implemented by means of a `dentry_hashtable` array. Each element is a pointer to a list of dentries that hash to the same hash table value. The array's size usually depends on the amount of RAM installed in the system; the default value is 256 entries per megabyte of RAM. The `d_hash` field of the dentry object contains pointers to the adjacent elements in the list associated with a single hash value. The hash function produces its value from both the dentry object of the directory and the filename.

The `dcache_lock` spin lock protects the dentry cache data structures against concurrent accesses in multiprocessor systems. The `d_lookup()` function looks in the hash table for a given parent dentry object and filename; to avoid race conditions, it makes use of a seqlock (see the section "Seqlocks" in Chapter 5). The `__d_lookup()` function is similar, but it assumes that no race condition can happen, so it does not use the seqlock.

Files Associated with a Process

We mentioned in the section "An Overview of the Unix Filesystem" in Chapter 1 that each process has its own current working directory and its own root directory. These are only two examples of data that must be maintained by the kernel to represent the interactions between a process and a filesystem. A whole data structure of type `fs_struct` is used for that purpose (see Table 12-6), and each process descriptor has an `fs` field that points to the process `fs_struct` structure.

Table 12-6. The fields of the fs_struct structure

Type	Field	Description
`atomic_t`	`count`	Number of processes sharing this table
`rwlock_t`	`lock`	Read/write spin lock for the table fields
`int`	`umask`	Bit mask used when opening the file to set the file permissions
`struct dentry *`	`root`	Dentry of the root directory
`struct dentry *`	`pwd`	Dentry of the current working directory
`struct dentry *`	`altroot`	Dentry of the emulated root directory (always NULL for the 80×86 architecture)

Table 12-6. The fields of the fs_struct structure (continued)

Type	Field	Description
struct vfsmount *	rootmnt	Mounted filesystem object of the root directory
struct vfsmount *	pwdmnt	Mounted filesystem object of the current working directory
struct vfsmount *	altrootmnt	Mounted filesystem object of the emulated root directory (always NULL for the 80 × 86 architecture)

A second table, whose address is contained in the `files` field of the process descriptor, specifies which files are currently opened by the process. It is a `files_struct` structure whose fields are illustrated in Table 12-7.

Table 12-7. The fields of the files_struct structure

Type	Field	Description
atomic_t	count	Number of processes sharing this table
rwlock_t	file_lock	Read/write spin lock for the table fields
int	max_fds	Current maximum number of file objects
int	max_fdset	Current maximum number of file descriptors
int	next_fd	Maximum file descriptors ever allocated plus 1
struct file **	fd	Pointer to array of file object pointers
fd_set *	close_on_exec	Pointer to file descriptors to be closed on exec( )
fd_set *	open_fds	Pointer to open file descriptors
fd_set	close_on_exec_init	Initial set of file descriptors to be closed on exec( )
fd_set	open_fds_init	Initial set of file descriptors
struct file *[]	fd_array	Initial array of file object pointers

The `fd` field points to an array of pointers to file objects. The size of the array is stored in the `max_fds` field. Usually, `fd` points to the `fd_array` field of the `files_struct` structure, which includes 32 file object pointers. If the process opens more than 32 files, the kernel allocates a new, larger array of file pointers and stores its address in the `fd` fields; it also updates the `max_fds` field.

For every file with an entry in the `fd` array, the array index is the *file descriptor*. Usually, the first element (index 0) of the array is associated with the standard input of the process, the second with the standard output, and the third with the standard error (see Figure 12-3). Unix processes use the file descriptor as the main file identifier. Notice that, thanks to the `dup( )`, `dup2( )`, and `fcntl( )` system calls, two file descriptors may refer to the same opened file—that is, two elements of the array could point to the same file object. Users see this all the time when they use shell constructs such as `2>&1` to redirect the standard error to the standard output.

A process cannot use more than `NR_OPEN` (usually, 1,048,576) file descriptors. The kernel also enforces a dynamic bound on the maximum number of file descriptors in

the signal->rlim[RLIMIT_NOFILE] structure of the process descriptor; this value is usually 1,024, but it can be raised if the process has root privileges.

The open_fds field initially contains the address of the open_fds_init field, which is a bitmap that identifies the file descriptors of currently opened files. The max_fdset field stores the number of bits in the bitmap. Because the fd_set data structure includes 1,024 bits, there is usually no need to expand the size of the bitmap. However, the kernel may dynamically expand the size of the bitmap if this turns out to be necessary, much as in the case of the array of file objects.
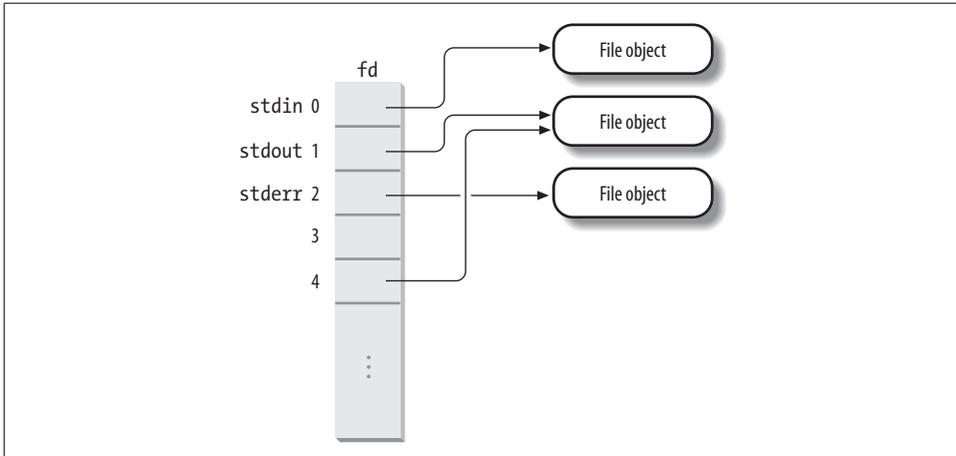
Figure 12-3. The fd array

The kernel provides an fget( ) function to be invoked when the kernel starts using a file object. This function receives as its parameter a file descriptor fd. It returns the address in current->files->fd[fd] (that is, the address of the corresponding file object), or NULL if no file corresponds to fd. In the first case, fget( ) increases the file object usage counter f_count by 1.

The kernel also provides an fput( ) function to be invoked when a kernel control path finishes using a file object. This function receives as its parameter the address of a file object and decreases its usage counter, f_count. Moreover, if this field becomes 0, the function invokes the release method of the file operations (if defined), decreases the i_writecount field in the inode object (if the file was opened for writing), removes the file object from the superblock's list, releases the file object to the slab allocator, and decreases the usage counters of the associated dentry object and of the filesystem descriptor (see the later section "Filesystem Mounting).

The fget_light() and fput_light() functions are faster versions of fget() and fput( ): the kernel uses them when it can safely assume that the current process already owns the file object—that is, the process has already previously increased the file object's reference counter. For instance, they are used by the service routines of the

system calls that receive a file descriptor as an argument, because the file object's reference counter has been increased by a previous open( ) system call.

Filesystem Types

The Linux kernel supports many different types of filesystems. In the following, we introduce a few special types of filesystems that play an important role in the internal design of the Linux kernel.

Next, we'll discuss filesystem registration—that is, the basic operation that must be performed, usually during system initialization, before using a filesystem type. Once a filesystem is registered, its specific functions are available to the kernel, so that type of filesystem can be mounted on the system's directory tree.

Special Filesystems

While network and disk-based filesystems enable the user to handle information stored outside the kernel, special filesystems may provide an easy way for system programs and administrators to manipulate the data structures of the kernel and to implement special features of the operating system. Table 12-8 lists the most common special filesystems used in Linux; for each of them, the table reports its suggested mount point and a short description.

Notice that a few filesystems have no fixed mount point (keyword "any" in the table). These filesystems can be freely mounted and used by the users. Moreover, some other special filesystems do not have a mount point at all (keyword "none" in the table). They are not for user interaction, but the kernel can use them to easily reuse some of the VFS layer code; for instance, we'll see in Chapter 19 that, thanks to the *pipefs* special filesystem, pipes can be treated in the same way as FIFO files.

Table 12-8. Most common special filesystems

Name	Mount point	Description
bdev	none	Block devices (see Chapter 13)
binfmt_misc	any	Miscellaneous executable formats (see Chapter 20)
devpts	/dev/pts	Pseudoterminal support (Open Group's Unix98 standard)
eventpollfs	none	Used by the efficient event polling mechanism
futexfs	none	Used by the futex (Fast Userspace Locking) mechanism
pipefs	none	Pipes (see Chapter 19)
proc	/proc	General access point to kernel data structures
rootfs	none	Provides an empty root directory for the bootstrap phase
shm	none	IPC-shared memory regions (see Chapter 19)
mqueue	any	Used to implement POSIX message queues (see Chapter 19)
sockfs	none	Sockets

Table 12-8. Most common special filesystems (continued)

Name	Mount point	Description
sysfs	*/sys*	General access point to system data (see Chapter 13)
tmpfs	any	Temporary files (kept in RAM unless swapped)
usbfs	*/proc/bus/usb*	USB devices

Special filesystems are not bound to physical block devices. However, the kernel assigns to each mounted special filesystem a fictitious block device that has the value 0 as major number and an arbitrary value (different for each special filesystem) as a minor number. The set_anon_super( ) function is used to initialize superblocks of special filesystems; this function essentially gets an unused minor number dev and sets the s_dev field of the new superblock with major number 0 and minor number dev. Another function called kill_anon_super( ) removes the superblock of a special filesystem. The unnamed_dev_idr variable includes pointers to auxiliary structures that record the minor numbers currently in use. Although some kernel designers dislike the fictitious block device identifiers, they help the kernel to handle special filesystems and regular ones in a uniform way.

We'll see a practical example of how the kernel defines and initializes a special filesystem in the later section "Mounting a Generic Filesystem."

Filesystem Type Registration

Often, the user configures Linux to recognize all the filesystems needed when compiling the kernel for his system. But the code for a filesystem actually may either be included in the kernel image or dynamically loaded as a module (see Appendix B). The VFS must keep track of all filesystem types whose code is currently included in the kernel. It does this by performing *filesystem type registration*.

Each registered filesystem is represented as a file_system_type object whose fields are illustrated in Table 12-9.

Table 12-9. The fields of the file_system_type object

Type	Field	Description
const char *	name	Filesystem name
int	fs_flags	Filesystem type flags
struct super_block * (*)()	get_sb	Method for reading a superblock
void (*)()	kill_sb	Method for removing a superblock
struct module *	owner	Pointer to the module implementing the filesystem (see Appendix B)
struct file_system_type *	next	Pointer to the next element in the list of filesystem types
struct list_head	fs_supers	Head of a list of superblock objects having the same filesystem type

All filesystem-type objects are inserted into a singly linked list. The `file_systems` variable points to the first item, while the `next` field of the structure points to the next item in the list. The `file_systems_lock` read/write spin lock protects the whole list against concurrent accesses.

The `fs_supers` field represents the head (first dummy element) of a list of superblock objects corresponding to mounted filesystems of the given type. The backward and forward links of a list element are stored in the `s_instances` field of the superblock object.

The `get_sb` field points to the filesystem-type-dependent function that allocates a new superblock object and initializes it (if necessary, by reading a disk). The `kill_sb` field points to the function that destroys a superblock.

The `fs_flags` field stores several flags, which are listed in Table 12-10.

Table 12-10. The filesystem type flags

Name	Description
FS_REQUIRES_DEV	Every filesystem of this type must be located on a physical disk device.
FS_BINARY_MOUNTDATA	The filesystem uses binary mount data.
FS_REVAL_DOT	Always revalidate the "." and ".." paths in the dentry cache (for network filesystems).
FS_ODD_RENAME	"Rename" operations are "move" operations (for network filesystems).

During system initialization, the `register_filesystem( )` function is invoked for every filesystem specified at compile time; the function inserts the corresponding `file_system_type` object into the filesystem-type list.

The `register_filesystem( )` function is also invoked when a module implementing a filesystem is loaded. In this case, the filesystem may also be unregistered (by invoking the `unregister_filesystem( )` function) when the module is unloaded.

The `get_fs_type( )` function, which receives a filesystem name as its parameter, scans the list of registered filesystems looking at the `name` field of their descriptors, and returns a pointer to the corresponding `file_system_type` object, if it is present.

Filesystem Handling

Like every traditional Unix system, Linux makes use of a *system's root filesystem*: it is the filesystem that is directly mounted by the kernel during the booting phase and that holds the system initialization scripts and the most essential system programs.

Other filesystems can be mounted—either by the initialization scripts or directly by the users—on directories of already mounted filesystems. Being a tree of directories, every filesystem has its own *root directory*. The directory on which a filesystem is mounted is called the *mount point*. A mounted filesystem is a *child* of the mounted

filesystem to which the mount point directory belongs. For instance, the */proc* virtual filesystem is a child of the system's root filesystem (and the system's root filesystem is the *parent* of */proc*). The root directory of a mounted filesystem hides the content of the mount point directory of the parent filesystem, as well as the whole subtree of the parent filesystem below the mount point.[*]

Namespaces

In a traditional Unix system, there is only one tree of mounted filesystems: starting from the system's root filesystem, each process can potentially access every file in a mounted filesystem by specifying the proper pathname. In this respect, Linux 2.6 is more refined: every process might have its own tree of mounted filesystems—the so-called *namespace* of the process.

Usually most processes share the same namespace, which is the tree of mounted filesystems that is rooted at the system's root filesystem and that is used by the *init* process. However, a process gets a new namespace if it is created by the clone() system call with the CLONE_NEWNS flag set (see the section "The clone(), fork(), and vfork() System Calls" in Chapter 3). The new namespace is then inherited by children processes if the parent creates them without the CLONE_NEWNS flag.

When a process mounts—or unmounts—a filesystem, it only modifies its namespace. Therefore, the change is visible to all processes that share the same namespace, and only to them. A process can even change the root filesystem of its namespace by using the Linux-specific pivot_root() system call.

The namespace of a process is represented by a namespace structure pointed to by the namespace field of the process descriptor. The fields of the namespace structure are shown in Table 12-11.

Table 12-11. The fields of the namespace structure

Type	Field	Description
atomic_t	count	Usage counter (how many processes share the namespace)
struct vfsmount *	root	Mounted filesystem descriptor for the root directory of the namespace
struct list_head	list	Head of list of all mounted filesystem descriptors
struct rw_semaphore	sem	Read/write semaphore protecting this structure

---

[*] The root directory of a filesystem can be different from the root directory of a process: as we have seen in the earlier section "Files Associated with a Process," the process's root directory is the directory corresponding to the "/" pathname. By default, the process' root directory coincides with the root directory of the system's root filesystem (or more precisely, with the root directory of the root filesystem in the namespace of the process, described in the following section), but it can be changed by invoking the chroot() system call.

The `list` field is the head of a doubly linked circular list collecting all mounted filesystems that belong to the namespace. The `root` field specifies the mounted filesystem that represents the root of the tree of mounted filesystems of this namespace. As we will see in the next section, mounted filesystems are represented by `vfsmount` structures.

Filesystem Mounting

In most traditional Unix-like kernels, each filesystem can be mounted only once. Suppose that an Ext2 filesystem stored in the */dev/fd0* floppy disk is mounted on */flp* by issuing the command:

```
mount -t ext2 /dev/fd0 /flp
```

Until the filesystem is unmounted by issuing a `umount` command, every other mount command acting on */dev/fd0* fails.

However, Linux is different: it is possible to mount the same filesystem several times. Of course, if a filesystem is mounted *n* times, its root directory can be accessed through *n* mount points, one per mount operation. Although the same filesystem can be accessed by using different mount points, it is really unique. Thus, there is only one superblock object for all of them, no matter of how many times it has been mounted.

Mounted filesystems form a hierarchy: the mount point of a filesystem might be a directory of a second filesystem, which in turn is already mounted over a third filesystem, and so on.[*]

It is also possible to stack multiple mounts on a single mount point. Each new mount on the same mount point hides the previously mounted filesystem, although processes already using the files and directories under the old mount can continue to do so. When the topmost mounting is removed, then the next lower mount is once more made visible.

As you can imagine, keeping track of mounted filesystems can quickly become a nightmare. For each mount operation, the kernel must save in memory the mount point and the mount flags, as well as the relationships between the filesystem to be mounted and the other mounted filesystems. Such information is stored in a *mounted filesystem descriptor* of type `vfsmount`. The fields of this descriptor are shown in Table 12-12.

---

[*] Quite surprisingly, the mount point of a filesystem might be a directory of the same filesystem, provided that it was already mounted. For instance:

```
mount -t ext2 /dev/fd0 /flp; touch /flp/foo
mkdir /flp/mnt; mount -t ext2 /dev/fd0 /flp/mnt
```

Now, the empty *foo* file on the floppy filesystem can be accessed both as */flp/foo* and */flp/mnt/foo*.

Table 12-12. The fields of the vfsmount data structure

Type	Field	Description
struct list_head	mnt_hash	Pointers for the hash table list.
struct vfsmount *	mnt_parent	Points to the parent filesystem on which this filesystem is mounted.
struct dentry *	mnt_mountpoint	Points to the dentry of the mount point directory where the filesystem is mounted.
struct dentry *	mnt_root	Points to the dentry of the root directory of this filesystem.
struct super_block *	mnt_sb	Points to the superblock object of this filesystem.
struct list_head	mnt_mounts	Head of a list including all filesystem descriptors mounted on directories of this filesystem.
struct list_head	mnt_child	Pointers for the mnt_mounts list of mounted filesystem descriptors.
atomic_t	mnt_count	Usage counter (increased to forbid filesystem unmounting).
int	mnt_flags	Flags.
int	mnt_expiry_mark	Flag set to true if the filesystem is marked as expired (the filesystem can be automatically unmounted if the flag is set and no one is using it).
char *	mnt_devname	Device filename.
struct list_head	mnt_list	Pointers for namespace's list of mounted filesystem descriptors.
struct list_head	mnt_fslink	Pointers for the filesystem-specific expire list.
struct namespace *	mnt_namespace	Pointer to the namespace of the process that mounted the filesystem.

The vfsmount data structures are kept in several doubly linked circular lists:

- A hash table indexed by the address of the vfsmount descriptor of the parent filesystem and the address of the dentry object of the mount point directory. The hash table is stored in the mount_hashtable array, whose size depends on the amount of RAM in the system. Each item of the table is the head of a circular doubly linked list storing all descriptors that have the same hash value. The mnt_hash field of the descriptor contains the pointers to adjacent elements in this list.

- For each namespace, a circular doubly linked list including all mounted filesystem descriptors belonging to the namespace. The list field of the namespace structure stores the head of the list, while the mnt_list field of the vfsmount descriptor contains the pointers to adjacent elements in the list.

- For each mounted filesystem, a circular doubly linked list including all child mounted filesystems. The head of each list is stored in the mnt_mounts field of the mounted filesystem descriptor; moreover, the mnt_child field of the descriptor stores the pointers to the adjacent elements in the list.

The vfsmount_lock spin lock protects the lists of mounted filesystem objects from concurrent accesses.

The mnt_flags field of the descriptor stores the value of several flags that specify how some kinds of files in the mounted filesystem are handled. These flags, which can be set through options of the *mount* command, are listed in Table 12-13.

Table 12-13. Mounted filesystem flags

Name	Description
MNT_NOSUID	Forbid setuid and setgid flags in the mounted filesystem
MNT_NODEV	Forbid access to device files in the mounted filesystem
MNT_NOEXEC	Disallow program execution in the mounted filesystem

Here are some functions that handle the mounted filesystem descriptors:

alloc_vfsmnt(name)
    Allocates and initializes a mounted filesystem descriptor

free_vfsmnt(mnt)
    Frees a mounted filesystem descriptor pointed to by mnt

lookup_mnt(mnt, dentry)
    Looks up a descriptor in the hash table and returns its address

Mounting a Generic Filesystem

We'll now describe the actions performed by the kernel in order to mount a filesystem. We'll start by considering a filesystem that is going to be mounted over a directory of an already mounted filesystem (in this discussion we will refer to this new filesystem as "generic").

The mount( ) system call is used to mount a generic filesystem; its sys_mount( ) service routine acts on the following parameters:

- The pathname of a device file containing the filesystem, or NULL if it is not required (for instance, when the filesystem to be mounted is network-based)
- The pathname of the directory on which the filesystem will be mounted (the mount point)
- The filesystem type, which must be the name of a registered filesystem
- The mount flags (permitted values are listed in Table 12-14)
- A pointer to a filesystem-dependent data structure (which may be NULL)

Table 12-14. Flags used by the mount() system call

Macro	Description
MS_RDONLY	Files can only be read
MS_NOSUID	Forbid setuid and setgid flags

Table 12-14. Flags used by the mount() system call (continued)

Macro	Description
MS_NODEV	Forbid access to device files
MS_NOEXEC	Disallow program execution
MS_SYNCHRONOUS	Write operations on files and directories are immediate
MS_REMOUNT	Remount the filesystem changing the mount flags
MS_MANDLOCK	Mandatory locking allowed
MS_DIRSYNC	Write operations on directories are immediate
MS_NOATIME	Do not update file access time
MS_NODIRATIME	Do not update directory access time
MS_BIND	Create a "bind mount," which allows making a file or directory visible at another point of the system directory tree (option `--bind` of the *mount* command)
MS_MOVE	Atomically move a mounted filesystem to another mount point (option `--move` of the *mount* command)
MS_REC	Recursively create "bind mounts" for a directory subtree
MS_VERBOSE	Generate kernel messages on mount errors

The sys_mount( ) function copies the value of the parameters into temporary kernel buffers, acquires the big kernel lock, and invokes the do_mount( ) function. Once do_mount( ) returns, the service routine releases the big kernel lock and frees the temporary kernel buffers.

The do_mount( ) function takes care of the actual mount operation by performing the following operations:

1. If some of the MS_NOSUID, MS_NODEV, or MS_NOEXEC mount flags are set, it clears them and sets the corresponding flag (MNT_NOSUID, MNT_NODEV, MNT_NOEXEC) in the mounted filesystem object.

2. Looks up the pathname of the mount point by invoking path_lookup( ); this function stores the result of the pathname lookup in the local variable nd of type nameidata (see the later section "Pathname Lookup").

3. Examines the mount flags to determine what has to be done. In particular:

    a. If the MS_REMOUNT flag is specified, the purpose is usually to change the mount flags in the s_flags field of the superblock object and the mounted filesystem flags in the mnt_flags field of the mounted filesystem object. The do_remount( ) function performs these changes.

    b. Otherwise, it checks the MS_BIND flag. If it is specified, the user is asking to make visible a file or directory on another point of the system directory tree.

    c. Otherwise, it checks the MS_MOVE flag. If it is specified, the user is asking to change the mount point of an already mounted filesystem. The do_move_mount( ) function does this atomically.

d. Otherwise, it invokes `do_new_mount()`. This is the most common case. It is triggered when the user asks to mount either a special filesystem or a regular filesystem stored in a disk partition. `do_new_mount()` invokes the `do_kern_mount()` function passing to it the filesystem type, the mount flags, and the block device name. This function, which takes care of the actual mount operation and returns the address of a new mounted filesystem descriptor, is described below. Next, `do_new_mount()` invokes `do_add_mount()`, which essentially performs the following actions:

1. Acquires for writing the `namespace->sem` semaphore of the current process, because the function is going to modify the namespace.

2. The `do_kern_mount()` function might put the current process to sleep; meanwhile, another process might mount a filesystem on the very same mount point as ours or even change our root filesystem (`current->namespace->root`). Verifies that the lastly mounted filesystem on this mount point still refers to the current's namespace; if not, releases the read/write semaphore and returns an error code.

3. If the filesystem to be mounted is already mounted on the mount point specified as parameter of the system call, or if the mount point is a symbolic link, it releases the read/write semaphore and returns an error code.

4. Initializes the flags in the `mnt_flags` field of the new mounted filesystem object allocated by `do_kern_mount()`.

5. Invokes `graft_tree()` to insert the new mounted filesystem object in the namespace list, in the hash table, and in the children list of the parent-mounted filesystem.

6. Releases the `namespace->sem` read/write semaphore and returns.

4. Invokes `path_release()` to terminate the pathname lookup of the mount point (see the later section "Pathname Lookup") and returns 0.

The do_kern_mount( ) function

The core of the mount operation is the `do_kern_mount()` function, which checks the filesystem type flags to determine how the mount operation is to be done. This function receives the following parameters:

`fstype`
   The name of the filesystem type to be mounted

`flags`
   The mount flags (see Table 12-14)

`name`
   The pathname of the block device storing the filesystem (or the filesystem type name for special filesystems)

data

  Pointer to additional data to be passed to the read_super method of the
  filesystem

The function takes care of the actual mount operation by performing essentially the
following operations:

1. Invokes get_fs_type() to search in the list of filesystem types and locate the
   name stored in the fstype parameter; get_fs_type() returns in the local variable
   type the address of the corresponding file_system_type descriptor.
2. Invokes alloc_vfsmnt() to allocate a new mounted filesystem descriptor and
   stores its address in the mnt local variable.
3. Invokes the type->get_sb() filesystem-dependent function to allocate a new
   superblock and to initialize it (see below).
4. Initializes the mnt->mnt_sb field with the address of the new superblock object.
5. Initializes the mnt->mnt_root field with the address of the dentry object corre-
   sponding to the root directory of the filesystem, and increases the usage counter
   of the dentry object.
6. Initializes the mnt->mnt_parent field with the value in mnt (for generic filesys-
   tems, the proper value of mnt_parent will be set when the mounted filesystem
   descriptor is inserted in the proper lists by graft_tree(); see step 3d5 of do_
   mount()).
7. Initializes the mnt->mnt_namespace field with the value in current->namespace.
8. Releases the s_umount read/write semaphore of the superblock object (it was
   acquired when the object was allocated in step 3).
9. Returns the address mnt of the mounted filesystem object.

Allocating a superblock object

The get_sb method of the filesystem object is usually implemented by a one-line
function. For instance, in the Ext2 filesystem the method is implemented as follows:

```
struct super_block * ext2_get_sb(struct file_system_type *type,
                      int flags, const char *dev_name, void *data)
{
    return get_sb_bdev(type, flags, dev_name, data, ext2_fill_super);
}
```

The get_sb_bdev() VFS function allocates and initializes a new superblock suitable
for disk-based filesystems; it receives the address of the ext2_fill_super() function,
which reads the disk superblock from the Ext2 disk partition.

To allocate superblocks suitable for special filesystems, the VFS also provides the
get_sb_pseudo() function (for special filesystems with no mount point such as
pipefs), the get_sb_single() function (for special filesystems with single mount point

such as *sysfs*), and the `get_sb_nodev()` function (for special filesystems that can be mounted several times such as *tmpfs*; see below).

The most important operations performed by `get_sb_bdev()` are the following:

1. Invokes `open_bdev_excl()` to open the block device having device file name `dev_name` (see the section "Character Device Drivers" in Chapter 13).

2. Invokes `sget()` to search the list of superblock objects of the filesystem (`type->fs_supers`, see the earlier section "Filesystem Type Registration"). If a superblock relative to the block device is already present, the function returns its address. Otherwise, it allocates and initializes a new superblock object, inserts it into the filesystem list and in the global list of superblocks, and returns its address.

3. If the superblock is not new (it was not allocated in the previous step, because the filesystem is already mounted), it jumps to step 6.

4. Copies the value of the `flags` parameter into the `s_flags` field of the superblock and sets the `s_id`, `s_old_blocksize`, and `s_blocksize` fields with the proper values for the block device.

5. Invokes the filesystem-dependent function passed as last argument to `get_sb_bdev()` to access the superblock information on disk and fill the other fields of the new superblock object.

6. Returns the address of the new superblock object.

Mounting the Root Filesystem

Mounting the root filesystem is a crucial part of system initialization. It is a fairly complex procedure, because the Linux kernel allows the root filesystem to be stored in many different places, such as a hard disk partition, a floppy disk, a remote filesystem shared via NFS, or even a *ramdisk* (a fictitious block device kept in RAM).

To keep the description simple, let's assume that the root filesystem is stored in a partition of a hard disk (the most common case, after all). While the system boots, the kernel finds the major number of the disk that contains the root filesystem in the `ROOT_DEV` variable (see Appendix A). The root filesystem can be specified as a device file in the */dev* directory either when compiling the kernel or by passing a suitable *"root"* option to the initial bootstrap loader. Similarly, the mount flags of the root filesystem are stored in the `root_mountflags` variable. The user specifies these flags either by using the *rdev* external program on a compiled kernel image or by passing a suitable *rootflags* option to the initial bootstrap loader (see Appendix A).

Mounting the root filesystem is a two-stage procedure, shown in the following list:

1. The kernel mounts the special *rootfs* filesystem, which simply provides an empty directory that serves as initial mount point.

2. The kernel mounts the real root filesystem over the empty directory.

Why does the kernel bother to mount the *rootfs* filesystem before the real one? Well, the *rootfs* filesystem allows the kernel to easily change the real root filesystem. In fact, in some cases, the kernel mounts and unmounts several root filesystems, one after the other. For instance, the initial bootstrap CD of a distribution might load in RAM a kernel with a minimal set of drivers, which mounts as root a minimal filesystem stored in a ramdisk. Next, the programs in this initial root filesystem probe the hardware of the system (for instance, they determine whether the hard disk is EIDE, SCSI, or whatever), load all needed kernel modules, and remount the root filesystem from a physical block device.

Phase 1: Mounting the rootfs filesystem

The first stage is performed by the `init_rootfs()` and `init_mount_tree()` functions, which are executed during system initialization.

The `init_rootfs()` function registers the special filesystem type *rootfs*:

```
struct file_system_type rootfs_fs_type = {
    .name = "rootfs";
    .get_sb = rootfs_get_sb;
    .kill_sb = kill_litter_super;
};
register_filesystem(&rootfs_fs_type);
```

The `init_mount_tree()` function executes the following operations:

1. Invokes `do_kern_mount()` passing to it the string `"rootfs"` as filesystem type, and stores the address of the mounted filesystem descriptor returned by this function in the `mnt` local variable. As explained in the previous section, `do_kern_mount()` ends up invoking the `get_sb` method of the *rootfs* filesystem, that is, the `rootfs_get_sb()` function:

   ```
   struct superblock *rootfs_get_sb(struct file_system_type *fs_type,
                       int flags, const char *dev_name, void *data)
   {
       return get_sb_nodev(fs_type, flags|MS_NOUSER, data,
                           ramfs_fill_super);
   }
   ```

   The `get_sb_nodev()` function, in turn, executes the following steps:

   a. Invokes `sget()` to allocate a new superblock passing as parameter the address of the `set_anon_super()` function (see the earlier section "Special Filesystems"). As a result, the `s_dev` field of the superblock is set in the appropriate way: major number 0, minor number different from those of other mounted special filesystems.

   b. Copies the value of the `flags` parameter into the `s_flags` field of the superblock.

   c. Invokes `ramfs_fill_super()` to allocate an inode object and a corresponding dentry object, and to fill the superblock fields. Because *rootfs* is a special

filesystem that has no disk superblock, only a couple of superblock operations need to be implemented.

    d. Returns the address of the new superblock.

2. Allocates a `namespace` object for the namespace of process 0, and inserts into it the mounted filesystem descriptor returned by `do_kern_mount()`:

```
namespace = kmalloc(sizeof(*namespace), GFP_KERNEL);
list_add(&mnt->mnt_list, &namespace->list);
namespace->root = mnt;
mnt->mnt_namespace = init_task.namespace = namespace;
```

3. Sets the `namespace` field of every other process in the system to the address of the namespace object; also initializes the `namespace->count` usage counter. (By default, all processes share the same, initial namespace.)

4. Sets the root directory and the current working directory of process 0 to the root filesystem.

Phase 2: Mounting the real root filesystem

The second stage of the mount operation for the root filesystem is performed by the kernel near the end of the system initialization. There are several ways to mount the real root filesystem, according to the options selected when the kernel has been compiled and to the boot options passed by the kernel loader. For the sake of brevity, we consider the case of a disk-based filesystem whose device file name has been passed to the kernel by means of the "*root*" boot parameter. We also assume that no initial special filesystem is used, except the *rootfs* filesystem.

The `prepare_namespace()` function executes the following operations:

1. Sets the `root_device_name` variable with the device filename obtained from the "*root*" boot parameter. Also, sets the `ROOT_DEV` variable with the major and minor numbers of the same device file.

2. Invokes the `mount_root()` function, which in turn:

    a. Invokes `sys_mknod()` (the service routine of the `mknod()` system call) to create a */dev/root* device file in the *rootfs* initial root filesystem, having the major and minor numbers as in `ROOT_DEV`.

    b. Allocates a buffer and fills it with a list of filesystem type names. This list is either passed to the kernel in the "*rootfstype*" boot parameter or built by scanning the elements in the singly linked list of filesystem types.

    c. Scans the list of filesystem type names built in the previous step. For each name, it invokes `sys_mount()` to try to mount the given filesystem type on the root device. Because each filesystem-specific method uses a different magic number, all `get_sb()` invocations will fail except the one that attempts to fill the superblock by using the function of the filesystem really used on the root device. The filesystem is mounted on a directory named */root* of the *rootfs* filesystem.

     d. Invokes `sys_chdir("/root")` to change the current directory of the process.

3. Moves the mount point of the mounted filesystem on the root directory of the *rootfs* filesystem:

```
sys_mount(".", "/", NULL, MS_MOVE, NULL);
sys_chroot(".");
```

Notice that the *rootfs* special filesystem is not unmounted: it is only hidden under the disk-based root filesystem.

Unmounting a Filesystem

The umount( ) system call is used to unmount a filesystem. The corresponding sys_ umount( ) service routine acts on two parameters: a filename (either a mount point directory or a block device filename) and a set of flags. It performs the following actions:

1. Invokes `path_lookup()` to look up the mount point pathname; this function returns the results of the lookup operation in a local variable `nd` of type `nameidata` (see next section).

2. If the resulting directory is not the mount point of a filesystem, it sets the `retval` return code to `-EINVAL` and jumps to step 6. This check is done by verifying that `nd->mnt->mnt_root` contains the address of the dentry object pointed to by `nd. dentry`.

3. If the filesystem to be unmounted has not been mounted in the namespace, it sets the `retval` return code to `-EINVAL` and jumps to step 6. (Recall that some special filesystems have no mount point.) This check is done by invoking the `check_mnt()` function on `nd->mnt`.

4. If the user does not have the privileges required to unmount the filesystem, it sets the `retval` return code to `-EPERM` and jumps to step 6.

5. Invokes `do_umount()` passing as parameters `nd.mnt` (the mounted filesystem object) and `flags` (the set of flags). This function performs essentially the following operations:

     a. Retrieves the address of the `sb` superblock object from the `mnt_sb` field of the mounted filesystem object.

     b. If the user asked to force the unmount operation, it interrupts any ongoing mount operation by invoking the `umount_begin` superblock operation.

     c. If the filesystem to be unmounted is the root filesystem and the user didn't ask to actually detach it, it invokes `do_remount_sb()` to remount the root filesystem read-only and terminates.

     d. Acquires for writing the `namespace->sem` read/write semaphore of the current process, and gets the `vfsmount_lock` spin lock.

     e. If the mounted filesystem does not include mount points for any child mounted filesystem, or if the user asked to forcibly detach the filesystem, it

invokes `umount_tree()` to unmount the filesystem (together with all children filesystems).

    f. Releases the `vfsmount_lock` spin lock and the `namespace->sem` read/write semaphore of the current process.

6. Decreases the usage counters of the dentry object corresponding to the root directory of the filesystem and of the mounted filesystem descriptor; these counters were increased by `path_lookup()`.

7. Returns the `retval` value.

Pathname Lookup

When a process must act on a file, it passes its file pathname to some VFS system call, such as `open()`, `mkdir()`, `rename()`, or `stat()`. In this section, we illustrate how the VFS performs a *pathname lookup*, that is, how it derives an inode from the corresponding file pathname.

The standard procedure for performing this task consists of analyzing the pathname and breaking it into a sequence of filenames. All filenames except the last must identify directories.

If the first character of the pathname is /, the pathname is absolute, and the search starts from the directory identified by `current->fs->root` (the process root directory). Otherwise, the pathname is relative, and the search starts from the directory identified by `current->fs->pwd` (the process-current directory).

Having in hand the dentry, and thus the inode, of the initial directory, the code examines the entry matching the first name to derive the corresponding inode. Then the directory file that has that inode is read from disk and the entry matching the second name is examined to derive the corresponding inode. This procedure is repeated for each name included in the path.

The dentry cache considerably speeds up the procedure, because it keeps the most recently used dentry objects in memory. As we saw before, each such object associates a filename in a specific directory to its corresponding inode. In many cases, therefore, the analysis of the pathname can avoid reading the intermediate directories from disk.

However, things are not as simple as they look, because the following Unix and VFS filesystem features must be taken into consideration:

- The access rights of each directory must be checked to verify whether the process is allowed to read the directory's content.
- A filename can be a symbolic link that corresponds to an arbitrary pathname; in this case, the analysis must be extended to all components of that pathname.

- Symbolic links may induce circular references; the kernel must take this possibility into account and break endless loops when they occur.

- A filename can be the mount point of a mounted filesystem. This situation must be detected, and the lookup operation must continue into the new filesystem.

- Pathname lookup has to be done inside the namespace of the process that issued the system call. The same pathname used by two processes with different namespaces may specify different files.

Pathname lookup is performed by the `path_lookup()` function, which receives three parameters:

name
> A pointer to the file pathname to be resolved.

flags
> The value of flags that represent how the looked-up file is going to be accessed. The allowed values are included later in Table 12-16.

nd
> The address of a `nameidata` data structure, which stores the results of the lookup operation and whose fields are shown in Table 12-15.

When `path_lookup()` returns, the `nameidata` structure pointed to by `nd` is filled with data pertaining to the pathname lookup operation.

Table 12-15. The fields of the nameidata data structure

Type	Field	Description
`struct dentry *`	`dentry`	Address of the dentry object
`struct vfs_mount *`	`mnt`	Address of the mounted filesystem object
`struct qstr`	`last`	Last component of the pathname (used when the LOOKUP_ PARENT flag is set)
`unsigned int`	`flags`	Lookup flags
`int`	`last_type`	Type of last component of the pathname (used when the LOOKUP_PARENT flag is set)
`unsigned int`	`depth`	Current level of symbolic link nesting (see below); it must be smaller than 6
`char[ ] *`	`saved_names`	Array of pathnames associated with nested symbolic links
`union`	`intent`	One-member union specifying how the file will be accessed

The `dentry` and `mnt` fields point respectively to the dentry object and the mounted filesystem object of the last resolved component in the pathname. These two fields "describe" the file that is identified by the given pathname.

Because the dentry object and the mounted filesystem object returned by the `path_lookup()` function in the `nameidata` structure represent the result of a lookup operation, both objects should not be freed until the caller of `path_lookup()` finishes using

them. Therefore, path_lookup() increases the usage counters of both objects. If the caller wants to release these objects, it invokes the path_release() function passing as parameter the address of a nameidata structure.

The flags field stores the value of some flags used in the lookup operation; they are listed in Table 12-16. Most of these flags can be set by the caller in the flags parameter of path_lookup().

Table 12-16. The flags of the lookup operation

Macro	Description
LOOKUP_FOLLOW	If the last component is a symbolic link, interpret (follow) it
LOOKUP_DIRECTORY	The last component must be a directory
LOOKUP_CONTINUE	There are still filenames to be examined in the pathname
LOOKUP_PARENT	Look up the directory that includes the last component of the pathname
LOOKUP_NOALT	Do not consider the emulated root directory (useless in the 80×86 architecture)
LOOKUP_OPEN	Intent is to open a file
LOOKUP_CREATE	Intent is to create a file (if it doesn't exist)
LOOKUP_ACCESS	Intent is to check user's permission for a file

The path_lookup() function executes the following steps:

1. Initializes some fields of the nd parameter as follows:

   a. Sets the last_type field to LAST_ROOT (this is needed if the pathname is a slash or a sequence of slashes; see the later section "Parent Pathname Lookup").

   b. Sets the flags field to the value of the flags parameter

   c. Sets the depth field to 0.

2. Acquires for reading the current->fs->lock read/write semaphore of the current process.

3. If the first character in the pathname is a slash (/), the lookup operation must start from the root directory of current: the function gets the addresses of the corresponding mounted filesystem object (current->fs->rootmnt) and dentry object (current->fs->root), increases their usage counters, and stores the addresses in nd->mnt and nd->dentry, respectively.

4. Otherwise, if the first character in the pathname is not a slash, the lookup operation must start from the current working directory of current: the function gets the addresses of the corresponding mounted filesystem object (current->fs->pwdmnt) and dentry object (current->fs->pwd), increases their usage counters, and stores the addresses in nd->mnt and nd->dentry, respectively.

5. Releases the current->fs->lock read/write semaphore of the current process.

6. Sets the total_link_count field in the descriptor of the current process to 0 (see the later section "Lookup of Symbolic Links").

7. Invokes the `link_path_walk()` function to take care of the undergoing lookup operation:

```
return link_path_walk(name, nd);
```

We are now ready to describe the core of the pathname lookup operation, namely the `link_path_walk()` function. It receives as its parameters a pointer `name` to the pathname to be resolved and the address `nd` of a `nameidata` data structure.

To make things a bit easier, we first describe what `link_path_walk()` does when `LOOKUP_PARENT` is not set and the pathname does not contain symbolic links (standard pathname lookup). Next, we discuss the case in which `LOOKUP_PARENT` is set: this type of lookup is required when creating, deleting, or renaming a directory entry, that is, during a parent pathname lookup. Finally, we explain how the function resolves symbolic links.

Standard Pathname Lookup

When the `LOOKUP_PARENT` flag is cleared, `link_path_walk()` performs the following steps.

1. Initializes the `lookup_flags` local variable with `nd->flags`.

2. Skips all leading slashes (/) before the first component of the pathname.

3. If the remaining pathname is empty, it returns the value 0. In the `nameidata` data structure, the `dentry` and `mnt` fields point to the objects relative to the last resolved component of the original pathname.

4. If the `depth` field of the `nd` descriptor is positive, it sets the `LOOKUP_FOLLOW` flag in the `lookup_flags` local variable (see the section "Lookup of Symbolic Links").

5. Executes a cycle that breaks the pathname passed in the `name` parameter into components (the intermediate slashes are treated as filename separators); for each component found, the function:

    a. Retrieves the address of the inode object of the last resolved component from `nd->dentry->d_inode`. (In the first iteration, the inode refers to the directory from where to start the pathname lookup.)

    b. Checks that the permissions of the last resolved component stored into the inode allow execution (in Unix, a directory can be traversed only if it is executable). If the inode has a custom `permission` method, the function executes it; otherwise, it executes the `exec_permission_lite()` function, which examines the access mode stored in the `i_mode` inode field and the privileges of the running process. In both cases, if the last resolved component does not allow execution, `link_path_walk()` breaks out of the cycle and returns an error code.

    c. Considers the next component to be resolved. From its name, the function computes a 32-bit hash value to be used when looking in the dentry cache hash table.

d. Skips any trailing slash (/) after the slash that terminates the name of the component to be resolved.

e. If the component to be resolved is the last one in the original pathname, it jumps to step 6.

f. If the name of the component is "." (a single dot), it continues with the next component ("." refers to the current directory, so it has no effect inside a pathname).

g. If the name of the component is ".." (two dots), it tries to climb to the parent directory:

   1. If the last resolved directory is the process's root directory (nd->dentry is equal to current->fs->root and nd->mnt is equal to current->fs->rootmnt), then climbing is not allowed: it invokes follow_mount() on the last resolved component (see below) and continues with the next component.

   2. If the last resolved directory is the root directory of the nd->mnt filesystem (nd->dentry is equal to nd->mnt->mnt_root) and the nd->mnt filesystem is not mounted on top of another filesystem (nd->mnt is equal to nd->mnt->mnt_parent), then the nd->mnt filesystem is usually[*] the namespace's root filesystem: in this case, climbing is impossible, thus invokes follow_mount() on the last resolved component (see below) and continues with the next component.

   3. If the last resolved directory is the root directory of the nd->mnt filesystem and the nd->mnt filesystem is mounted on top of another filesystem, a filesystem switch is required. So, the function sets nd->dentry to nd->mnt->mnt_mountpoint, and nd->mnt to nd->mnt->mnt_parent, then restarts step 5g (recall that several filesystems can be mounted on the same mount point).

   4. If the last resolved directory is not the root directory of a mounted filesystem, then the function must simply climb to the parent directory: it sets nd->dentry to nd->dentry->d_parent, invokes follow_mount() on the parent directory, and continues with the next component.

The follow_mount() function checks whether nd->dentry is a mount point for some filesystem (nd->dentry->d_mounted is greater than zero); in this case, it invokes lookup_mnt() to search the root directory of the mounted filesystem in the dentry cache, and updates nd->dentry and nd->mnt with the object addresses corresponding to the mounted filesystem; then, it repeats the whole operation (there can be several filesystems mounted on the same mount point). Essentially, invoking the follow_mount() function when

---

[*] This case can also occur for network filesystems disconnected from the namespace's directory tree.

climbing to the parent directory is required because the process could start the pathname lookup from a directory included in a filesystem hidden by another filesystem mounted over the parent directory.

h. The component name is neither "." nor "..", so the function must look it up in the dentry cache. If the low-level filesystem has a custom `d_hash` dentry method, the function invokes it to modify the hash value already computed in step 5c.

i. Sets the `LOOKUP_CONTINUE` flag in `nd->flags` to denote that there is a next component to be analyzed.

j. Invokes `do_lookup()` to derive the dentry object associated with a given parent directory (`nd->dentry`) and filename (the pathname component being resolved). The function essentially invokes `__d_lookup()` first to search the dentry object of the component in the dentry cache. If no such object exists, `do_lookup()` invokes `real_lookup()`. This latter function reads the directory from disk by executing the `lookup` method of the inode, creates a new dentry object and inserts it in the dentry cache, then creates a new inode object and inserts it into the inode cache.[*] At the end of this step, the `dentry` and `mnt` fields of the `next` local variable will point, respectively, to the dentry object and the mounted filesystem object of the component name to be resolved in this cycle.

k. Invokes the `follow_mount()` function to check whether the component just resolved (`next.dentry`) refers to a directory that is a mount point for some filesystem (`next.dentry->d_mounted` is greater than zero). `follow_mount()` updates `next.dentry` and `next.mnt` so that they point to the dentry object and mounted filesystem object of the upmost filesystem mounted on the directory specified by this pathname component (see step 5g).

l. Checks whether the component just resolved refers to a symbolic link (`next.dentry->d_inode` has a custom `follow_link` method). We'll deal with this case in the later section "Lookup of Symbolic Links."

m. Checks whether the component just resolved refers to a directory (`next.dentry->d_inode` has a custom `lookup` method). If not, returns the error `-ENOTDIR`, because the component is in the middle of the original pathname.

n. Sets `nd->dentry` to `next.dentry` and `nd->mnt` to `next.mnt`, then continues with the next component of the pathname.

6. Now all components of the original pathname are resolved except the last one. Clears the `LOOKUP_CONTINUE` flag in `nd->flags`.

---

[*] In a few cases, the function might find the required inode already in the inode cache. This happens when the pathname component is the last one and it does not refer to a directory, the corresponding file has several hard links, and finally the file has been recently accessed through a hard link different from the one used in this pathname.

7. If the pathname has a trailing slash, it sets the `LOOKUP_FOLLOW` and `LOOKUP_DIRECTORY` flags in the `lookup_flags` local variable to force the last component to be interpreted by later functions as a directory name.

8. Checks the value of the `LOOKUP_PARENT` flag in the `lookup_flags` variable. In the following, we assume that the flag is set to 0, and we postpone the opposite case to the next section.

9. If the name of the last component is "." (a single dot), terminates the execution and returns the value 0 (no error). In the `nameidata` structure that `nd` points to, the `dentry` and `mnt` fields refer to the objects relative to the next-to-last component of the pathname (each component "." has no effect inside a pathname).

10. If the name of the last component is ".." (two dots), it tries to climb to the parent directory:

    a. If the last resolved directory is the process's root directory (`nd->dentry` is equal to `current->fs->root` and `nd->mnt` is equal to `current->fs->rootmnt`), it invokes `follow_mount()` on the next-to-last component and terminates the execution and returns the value 0 (no error). `nd->dentry` and `nd->mnt` refer to the objects relative to the next-to-last component of the pathname—that is, to the root directory of the process.

    b. If the last resolved directory is the root directory of the `nd->mnt` filesystem (`nd->dentry` is equal to `nd->mnt->mnt_root`) and the `nd->mnt` filesystem is not mounted on top of another filesystem (`nd->mnt` is equal to `nd->mnt->mnt_parent`), then climbing is impossible, thus invokes `follow_mount()` on the next-to-last component and terminates the execution and returns the value 0 (no error).

    c. If the last resolved directory is the root directory of the `nd->mnt` filesystem and the `nd->mnt` filesystem is mounted on top of another filesystem, it sets `nd->dentry` to `nd->mnt->mnt_mountpoint` and `nd->mnt` to `nd->mnt->mnt_parent`, then restarts step 10.

    d. If the last resolved directory is not the root directory of a mounted filesystem, it sets `nd->dentry` to `nd->dentry->d_parent`, invokes `follow_mount()` on the parent directory, and terminates the execution and returns the value 0 (no error). `nd->dentry` and `nd->mnt` refer to the objects relative to the component preceding the next-to-last component of the pathname.

11. The name of the last component is neither "." nor "..", so the function must look it up in the dentry cache. If the low-level filesystem has a custom `d_hash` dentry method, the function invokes it to modify the hash value already computed in step 5c.

12. Invokes `do_lookup()` to derive the dentry object associated with the parent directory and the filename (see step 5j). At the end of this step, the `next` local variable contains the pointers to both the dentry and the mounted filesystem descriptor relative to the last component name.

13. Invokes `follow_mount()` to check whether the last component is a mount point for some filesystem and, if this is the case, to update the `next` local variable with the addresses of the dentry object and mounted filesystem object relative to the root directory of the upmost mounted filesystem.

14. Checks whether the `LOOKUP_FOLLOW` flag is set in `lookup_flags` and the inode object `next.dentry->d_inode` has a custom `follow_link` method. If this is the case, the component is a symbolic link that must be interpreted, as described in the later section "Lookup of Symbolic Links."

15. The component is not a symbolic link or the symbolic link should not be interpreted. Sets the `nd->mnt` and `nd->dentry` fields with the value stored in `next.mnt` and `next.dentry`, respectively. The final dentry object is the result of the whole lookup operation.

16. Checks whether `nd->dentry->d_inode` is `NULL`. This happens when there is no inode associated with the dentry object, usually because the pathname refers to a nonexistent file. In this case, the function returns the error code `-ENOENT`.

17. There is an inode associated with the last component of the pathname. If the `LOOKUP_DIRECTORY` flag is set in `lookup_flags`, it checks that the inode has a custom `lookup` method—that is, it is a directory. If not, the function returns the error code `-ENOTDIR`.

18. Returns the value 0 (no error). `nd->dentry` and `nd->mnt` refer to the last component of the pathname.

Parent Pathname Lookup

In many cases, the real target of a lookup operation is not the last component of the pathname, but the next-to-last one. For example, when a file is created, the last component denotes the filename of the not yet existing file, and the rest of the pathname specifies the directory in which the new link must be inserted. Therefore, the lookup operation should fetch the dentry object of the next-to-last component. For another example, unlinking a file identified by the pathname */foo/bar* consists of removing *bar* from the directory *foo*. Thus, the kernel is really interested in accessing the directory *foo* rather than *bar*.

The `LOOKUP_PARENT` flag is used whenever the lookup operation must resolve the directory containing the last component of the pathname, rather than the last component itself.

When the `LOOKUP_PARENT` flag is set, the `link_path_walk()` function also sets up the `last` and `last_type` fields of the `nameidata` data structure. The `last` field stores the name of the last component in the pathname. The `last_type` field identifies the type of the last component; it may be set to one of the values shown in Table 12-17.

Table 12-17. The values of the last_type field in the nameidata data structure

Value	Description
LAST_NORM	Last component is a regular filename
LAST_ROOT	Last component is "/" (that is, the entire pathname is "/")
LAST_DOT	Last component is "."
LAST_DOTDOT	Last component is ".."
LAST_BIND	Last component is a symbolic link into a special filesystem

The LAST_ROOT flag is the default value set by path_lookup( ) when the whole pathname lookup operation starts (see the description at the beginning of the section "Pathname Lookup"). If the pathname turns out to be simply "/", the kernel does not change the initial value of the last_type field.

The remaining values of the last_type field are set by link_path_walk( ) when the LOOKUP_PARENT flag is set; in this case, the function performs the same steps described in the previous section up to step 8. From step 8 onward, however, the lookup operation for the last component of the pathname is different:

1. Sets nd->last to the name of the last component.

2. Initializes nd->last_type to LAST_NORM.

3. If the name of the last component is "." (a single dot), it sets nd->last_type to LAST_DOT.

4. If the name of the last component is ".." (two dots), it sets nd->last_type to LAST_DOTDOT.

5. Returns the value 0 (no error).

As you can see, the last component is not interpreted at all. Thus, when the function terminates, the dentry and mnt fields of the nameidata data structure point to the objects relative to the directory that includes the last component.

Lookup of Symbolic Links

Recall that a symbolic link is a regular file that stores a pathname of another file. A pathname may include symbolic links, and they must be resolved by the kernel.

For example, if */foo/bar* is a symbolic link pointing to (containing the pathname) *../dir*, the pathname */foo/bar/file* must be resolved by the kernel as a reference to the file */dir/file*. In this example, the kernel must perform two different lookup operations. The first one resolves */foo/bar*: when the kernel discovers that *bar* is the name of a symbolic link, it must retrieve its content and interpret it as another pathname. The second pathname operation starts from the directory reached by the first operation and continues until the last component of the symbolic link pathname has been resolved. Next, the original lookup operation resumes from the dentry reached in the second one and with the component following the symbolic link in the original pathname.

To further complicate the scenario, the pathname included in a symbolic link may include other symbolic links. You might think that the kernel code that resolves the symbolic links is hard to understand, but this is not true; the code is actually quite simple because it is recursive.

However, untamed recursion is intrinsically dangerous. For instance, suppose that a symbolic link points to itself. Of course, resolving a pathname including such a symbolic link may induce an endless stream of recursive invocations, which in turn quickly leads to a kernel stack overflow. The link_count field in the descriptor of the current process is used to avoid the problem: the field is increased before each recursive execution and decreased right after. If a sixth nested lookup operation is attempted, the whole lookup operation terminates with an error code. Therefore, the level of nesting of symbolic links can be at most 5.

Furthermore, the total_link_count field in the descriptor of the current process keeps track of how many symbolic links (even nonnested) were followed in the original lookup operation. If this counter reaches the value 40, the lookup operation aborts. Without this counter, a malicious user could create a pathological pathname including many consecutive symbolic links that freeze the kernel in a very long lookup operation.

This is how the code basically works: once the link_path_walk() function retrieves the dentry object associated with a component of the pathname, it checks whether the corresponding inode object has a custom follow_link method (see step 5l and step 14 in the section "Standard Pathname Lookup"). If so, the inode is a symbolic link that must be interpreted before proceeding with the lookup operation of the original pathname.

In this case, the link_path_walk() function invokes do_follow_link(), passing to it the address dentry of the dentry object of the symbolic link and the address nd of the nameidata data structure. In turn, do_follow_link() performs the following steps:

1. Checks that current->link_count is less than 5; otherwise, it returns the error code -ELOOP.

2. Checks that current->total_link_count is less than 40; otherwise, it returns the error code -ELOOP.

3. Invokes cond_resched() to perform a process switch if required by the current process (flag TIF_NEED_RESCHED in the thread_info descriptor of the current process set).

4. Increases current->link_count, current->total_link_count, and nd->depth.

5. Updates the access time of the inode object associated with the symbolic link to be resolved.

6. Invokes the filesystem-dependent function that implements the follow_link method passing to it the dentry and nd parameters. This function extracts the pathname stored in the symbolic link's inode, and saves this pathname in the proper entry of the nd->saved_names array.

7. Invokes the `__vfs_follow_link()` function passing to it the address `nd` and the address of the pathname in the `nd->saved_names` array (see below).

8. If defined, executes the `put_link` method of the inode object, thus releasing the temporary data structures allocated by the `follow_link` method.

9. Decreases the `current->link_count` and `nd->depth` fields.

10. Returns the error code returned by the `__vfs_follow_link()` function (0 for no error).

In turn, the `__vfs_follow_link()` does essentially the following:

1. Checks whether the first character of the pathname stored in the symbolic link is a slash: in this case an absolute pathname has been found, so there is no need to keep in memory any information about the previous path. If so, invokes `path_release()` on the `nameidata` structure, thus releasing the objects resulting from the previous lookup steps; then, the function sets the `dentry` and `mnt` fields of the `nameidata` data structure to the current process root directory.

2. Invokes `link_path_walk()` to resolve the symbolic link pathname, passing to it as parameters the pathname and `nd`.

3. Returns the value taken from `link_path_walk()`.

When `do_follow_link()` finally terminates, it has set the `dentry` field of the `next` local variable with the address of the dentry object referred to by the symbolic link to the original execution of `link_path_walk()`. The `link_path_walk()` function can then proceed with the next step.

Implementations of VFS System Calls

For the sake of brevity, we cannot discuss the implementation of all the VFS system calls listed in Table 12-1. However, it could be useful to sketch out the implementation of a few system calls, in order to show how VFS's data structures interact.

Let's reconsider the example proposed at the beginning of this chapter: a user issues a shell command that copies the MS-DOS file */floppy/TEST* to the Ext2 file */tmp/test*. The command shell invokes an external program such as *cp*, which we assume executes the following code fragment:

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test", O_WRONLY | O_CREAT | O_TRUNC, 0600);
do {
    len = read(inf, buf, 4096);
    write(outf, buf, len);
} while (len);
close(outf);
close(inf);
```

Actually, the code of the real *cp* program is more complicated, because it must also check for possible error codes returned by each system call. In our example, we focus our attention on the "normal" behavior of a copy operation.

The open( ) System Call

The open( ) system call is serviced by the sys_open( ) function, which receives as its parameters the pathname `filename` of the file to be opened, some access mode flags `flags`, and a permission bit mask `mode` if the file must be created. If the system call succeeds, it returns a file descriptor—that is, the index assigned to the new file in the current->files->fd array of pointers to file objects; otherwise, it returns −1.

In our example, open( ) is invoked twice; the first time to open */floppy/TEST* for reading (O_RDONLY flag) and the second time to open */tmp/test* for writing (O_WRONLY flag). If */tmp/test* does not already exist, it is created (O_CREAT flag) with exclusive read and write access for the owner (octal 0600 number in the third parameter).

Conversely, if the file already exists, it is rewritten from scratch (O_TRUNC flag). Table 12-18 lists all flags of the open( ) system call.

Table 12-18. The flags of the open( ) system call

Flag name	Description
O_RDONLY	Open for reading
O_WRONLY	Open for writing
O_RDWR	Open for both reading and writing
O_CREAT	Create the file if it does not exist
O_EXCL	With O_CREAT, fail if the file already exists
O_NOCTTY	Never consider the file as a controlling terminal
O_TRUNC	Truncate the file (remove all existing contents)
O_APPEND	Always write at end of the file
O_NONBLOCK	No system calls will block on the file
O_NDELAY	Same as O_NONBLOCK
O_SYNC	Synchronous write (block until physical write terminates)
FASYNC	I/O event notification via signals
O_DIRECT	Direct I/O transfer (no kernel buffering)
O_LARGEFILE	Large file (size greater than 2 GB)
O_DIRECTORY	Fail if file is not a directory
O_NOFOLLOW	Do not follow a trailing symbolic link in pathname
O_NOATIME	Do not update the inode's last access time

Let's describe the operation of the sys_open( ) function. It performs the following steps:

1. Invokes `getname()` to read the file pathname from the process address space.

2. Invokes `get_unused_fd()` to find an empty slot in `current->files->fd`. The corresponding index (the new file descriptor) is stored in the `fd` local variable.

3. Invokes the `filp_open()` function, passing as parameters the pathname, the access mode flags, and the permission bit mask. This function, in turn, executes the following steps:

    a. Copies the access mode flags into `namei_flags`, but encodes the access mode flags `O_RDONLY`, `O_WRONLY`, and `O_RDWR` with a special format: the bit at index 0 (lowest-order) of `namei_flags` is set only if the file access requires read privileges; similarly, the bit at index 1 is set only if the file access requires write privileges. Notice that it is not possible to specify in the `open()` system call that a file access does not require either read or write privileges; this makes sense, however, in a pathname lookup operation involving symbolic links.

    b. Invokes `open_namei()`, passing to it the pathname, the modified access mode flags, and the address of a local `nameidata` data structure. The function performs the lookup operation in the following manner:

        - If `O_CREAT` is not set in the access mode flags, starts the lookup operation with the `LOOKUP_PARENT` flag not set and the `LOOKUP_OPEN` flag set. Moreover, the `LOOKUP_FOLLOW` flag is set only if `O_NOFOLLOW` is cleared, while the `LOOKUP_DIRECTORY` flag is set only if the `O_DIRECTORY` flag is set.

        - If `O_CREAT` is set in the access mode flags, starts the lookup operation with the `LOOKUP_PARENT`, `LOOKUP_OPEN`, and `LOOKUP_CREATE` flags set. Once the `path_lookup()` function successfully returns, checks whether the requested file already exists. If not, allocates a new disk inode by invoking the `create` method of the parent inode.

    The `open_namei()` function also executes several security checks on the file located by the lookup operation. For instance, the function checks whether the inode associated with the dentry object found really exists, whether it is a regular file, and whether the current process is allowed to access it according to the access mode flags. Also, if the file is opened for writing, the function checks that the file is not locked by other processes.

    c. Invokes the `dentry_open()` function, passing to it the addresses of the dentry object and the mounted filesystem object located by the lookup operation, and the access mode flags. In turn, this function:

        1. Allocates a new file object.

        2. Initializes the `f_flags` and `f_mode` fields of the file object according to the access mode flags passed to the `open()` system call.

        3. Initializes the `f_dentry` and `f_vfsmnt` fields of the file object according to the addresses of the dentry object and the mounted filesystem object passed as parameters.

4. Sets the `f_op` field to the contents of the `i_fop` field of the corresponding inode object. This sets up all the methods for future file operations.

5. Inserts the file object into the list of opened files pointed to by the `s_files` field of the filesystem's superblock.

6. If the `open` method of the file operations is defined, the function invokes it.

7. Invokes `file_ra_state_init()` to initialize the read-ahead data structures (see Chapter 16).

8. If the `O_DIRECT` flag is set, it checks whether direct I/O operations can be performed on the file (see Chapter 16).

9. Returns the address of the file object.

d. Returns the address of the file object.

4. Sets `current->files->fd[fd]` to the address of the file object returned by `dentry_open()`.

5. Returns `fd`.

The read( ) and write( ) System Calls

Let's return to the code in our *cp* example. The `open( )` system calls return two file descriptors, which are stored in the `inf` and `outf` variables. Then the program starts a loop: at each iteration, a portion of the */floppy/TEST* file is copied into a local buffer (`read( )` system call), and then the data in the local buffer is written into the */tmp/test* file (`write( )` system call).

The `read( )` and `write( )` system calls are quite similar. Both require three parameters: a file descriptor `fd`, the address `buf` of a memory area (the buffer containing the data to be transferred), and a number `count` that specifies how many bytes should be transferred. Of course, `read( )` transfers the data from the file into the buffer, while `write( )` does the opposite. Both system calls return either the number of bytes that were successfully transferred or –1 to signal an error condition.

A return value less than `count` does not mean that an error occurred. The kernel is always allowed to terminate the system call even if not all requested bytes were transferred, and the user application must accordingly check the return value and reissue, if necessary, the system call. Typically, a small value is returned when reading from a pipe or a terminal device, when reading past the end of the file, or when the system call is interrupted by a signal. The end-of-file condition (EOF) can easily be recognized by a zero return value from `read( )`. This condition will not be confused with an abnormal termination due to a signal, because if `read( )` is interrupted by a signal before a data is read, an error occurs.

The read or write operation always takes place at the file offset specified by the current file pointer (field `f_pos` of the file object). Both system calls update the file pointer by adding the number of transferred bytes to it.

In short, both sys_read() (the read()'s service routine) and sys_write() (the write()'s service routine) perform almost the same steps:

1. Invokes fget_light() to derive from fd the address file of the corresponding file object (see the earlier section "Files Associated with a Process").

2. If the flags in file->f_mode do not allow the requested access (read or write operation), it returns the error code -EBADF.

3. If the file object does not have a read() or aio_read() (write() or aio_write()) file operation, it returns the error code -EINVAL.

4. Invokes access_ok() to perform a coarse check on the buf and count parameters (see the section "Verifying the Parameters" in Chapter 10).

5. Invokes rw_verify_area() to check whether there are conflicting mandatory locks for the file portion to be accessed. If so, it returns an error code, or puts the current process to sleep if the lock has been requested with a F_SETLKW command (see the section "File Locking" later in this chapter).

6. If defined, it invokes either the file->f_op->read or file->f_op->write method to transfer the data; otherwise, invokes either the file->f_op->aio_read or file->f_op->aio_write method. All these methods, which are discussed in Chapter 16, return the number of bytes that were actually transferred. As a side effect, the file pointer is properly updated.

7. Invokes fput_light() to release the file object.

8. Returns the number of bytes actually transferred.

The close( ) System Call

The loop in our example code terminates when the read() system call returns the value 0—that is, when all bytes of */floppy/TEST* have been copied into */tmp/test*. The program can then close the open files, because the copy operation has completed.

The close( ) system call receives as its parameter fd, which is the file descriptor of the file to be closed. The sys_close( ) service routine performs the following operations:

1. Gets the file object address stored in current->files->fd[fd]; if it is NULL, returns an error code.

2. Sets current->files->fd[fd] to NULL. Releases the file descriptor fd by clearing the corresponding bits in the open_fds and close_on_exec fields of current->files (see Chapter 20 for the Close on Execution flag).

3. Invokes filp_close( ), which performs the following operations:

   a. Invokes the flush method of the file operations, if defined.

   b. Releases all mandatory locks on the file, if any (see next section).

   c. Invokes fput( ) to release the file object.

4. Returns 0 or an error code. An error code can be raised by the flush method or by an error in a previous write operation on the file.

File Locking

When a file can be accessed by more than one process, a synchronization problem occurs. What happens if two processes try to write in the same file location? Or again, what happens if a process reads from a file location while another process is writing into it?

In traditional Unix systems, concurrent accesses to the same file location produce unpredictable results. However, Unix systems provide a mechanism that allows the processes to *lock* a file region so that concurrent accesses may be easily avoided.

The POSIX standard requires a file-locking mechanism based on the `fcntl()` system call. It is possible to lock an arbitrary region of a file (even a single byte) or to lock the whole file (including data appended in the future). Because a process can choose to lock only a part of a file, it can also hold multiple locks on different parts of the file.

This kind of lock does not keep out another process that is ignorant of locking. Like a semaphore used to protect a critical region in code, the lock is considered "advisory" because it doesn't work unless other processes cooperate in checking the existence of a lock before accessing the file. Therefore, POSIX's locks are known as *advisory locks*.

Traditional BSD variants implement advisory locking through the `flock()` system call. This call does not allow a process to lock a file region, only the whole file. Traditional System V variants provide the `lockf()` library function, which is simply an interface to `fcntl()`.

More importantly, System V Release 3 introduced *mandatory locking*: the kernel checks that every invocation of the `open()`, `read()`, and `write()` system calls does not violate a mandatory lock on the file being accessed. Therefore, mandatory locks are enforced even between noncooperative processes.[*]

Whether processes use advisory or mandatory locks, they can use both shared *read locks* and exclusive *write locks*. Several processes may have read locks on some file region, but only one process can have a write lock on it at the same time. Moreover, it is not possible to get a write lock when another process owns a read lock for the same file region, and vice versa.

---

[*] Oddly enough, a process may still unlink (delete) a file even if some other process owns a mandatory lock on it! This perplexing situation is possible because when a process deletes a file hard link, it does not modify its contents, but only the contents of its parent directory.

Linux File Locking

Linux supports all types of file locking: advisory and mandatory locks, plus the `fcntl()` and `flock()` system calls (`lockf()` is implemented as a standard library function).

The expected behavior of the `flock()` system call in every Unix-like operating system is to produce advisory locks only, without regard for the `MS_MANDLOCK` mount flag. In Linux, however, a special kind of `flock()`'s mandatory lock is used to support some proprietary network filesystems. It is the so-called *share-mode mandatory lock*; when set, no other process may open a file that would conflict with the access mode of the lock. Use of this feature for native Unix applications is discouraged, because the resulting source code will be nonportable.

Another kind of `fcntl()`-based mandatory lock called *lease* has been introduced in Linux. When a process tries to open a file protected by a lease, it is blocked as usual. However, the process that owns the lock receives a signal. Once informed, it should first update the file so that its content is consistent, and then release the lock. If the owner does not do this in a well-defined time interval (tunable by writing a number of seconds into */proc/sys/fs/lease-break-time*, usually 45 seconds), the lease is automatically removed by the kernel and the blocked process is allowed to continue.

A process can get or release an advisory file lock on a file in two possible ways:

- By issuing the `flock()` system call. The two parameters of the system call are the `fd` file descriptor, and a command to specify the lock operation. The lock applies to the whole file.

- By using the `fcntl()` system call. The three parameters of the system call are the `fd` file descriptor, a command to specify the lock operation, and a pointer to a `flock` structure (see Table 12-20). A couple of fields in this structure allow the process to specify the portion of the file to be locked. Processes can thus hold several locks on different portions of the same file.

Both the `fcntl()` and the `flock()` system call may be used on the same file at the same time, but a file locked through `fcntl()` does not appear locked to `flock()`, and vice versa. This has been done on purpose in order to avoid the deadlocks occurring when an application using a type of lock relies on a library that uses the other type.

Handling mandatory file locks is a bit more complex. Here are the steps to follow:

1. Mount the filesystem where mandatory locking is required using the `-o mand` option in the *mount* command, which sets the `MS_MANDLOCK` flag in the `mount()` system call. The default is to disable mandatory locking.

2. Mark the files as candidates for mandatory locking by setting their set-group bit (SGID) and clearing the group-execute permission bit. Because the set-group bit makes no sense when the group-execute bit is off, the kernel interprets that combination as a hint to use mandatory locks instead of advisory ones.

3. Uses the `fcntl()` system call (see below) to get or release a file lock.

Handling leases is much simpler than handling mandatory locks: it is sufficient to invoke a fcntl() system call with a F_SETLEASE or F_GETLEASE command. Another fcntl() invocation with the F_SETSIG command may be used to change the type of signal to be sent to the lease process holder.

Besides the checks in the read() and write() system calls, the kernel takes into consideration the existence of mandatory locks when servicing all system calls that could modify the contents of a file. For instance, an open() system call with the O_TRUNC flag set fails if any mandatory lock exists for the file.

The following section describes the main data structure used by the kernel to handle file locks issued by means of the flock() system call (FL_FLOCK locks) and of the fcntl() system call (FL_POSIX locks).

File-Locking Data Structures

All type of Linux locks are represented by the same file_lock data structure whose fields are shown in Table 12-19.

Table 12-19. The fields of the file_lock data structure

Type	Field	Description
struct file_lock *	fl_next	Next element in list of locks associated with the inode
struct list_head	fl_link	Pointers for active or blocked list
struct list_head	fl_block	Pointers for the lock's waiters list
struct files_struct *	fl_owner	Owner's files_struct
unsigned int	fl_pid	PID of the process owner
wait_queue_head_t	fl_wait	Wait queue of blocked processes
struct file *	fl_file	Pointer to file object
unsigned char	fl_flags	Lock flags
unsigned char	fl_type	Lock type
loff_t	fl_start	Starting offset of locked region
loff_t	fl_end	Ending offset of locked region
struct fasync_struct *	fl_fasync	Used for lease break notifications
unsigned long	fl_break_time	Remaining time before end of lease
struct file_lock_operations *	fl_ops	Pointer to file lock operations
struct lock_manager_operations *	fl_mops	Pointer to lock manager operations
union	fl_u	Filesystem-specific information

All lock_file structures that refer to the same file on disk are collected in a singly linked list, whose first element is pointed to by the i_flock field of the inode object. The fl_next field of the lock_file structure specifies the next element in the list.

When a process issues a blocking system call to require an exclusive lock while there are shared locks on the same file, the lock request cannot be satisfied immediately and the process must be suspended. The process is thus inserted into a wait queue pointed to by the fl_wait field of the blocked lock's file_lock structure. Two lists are used to distinguish lock requests that have been satisfied (*active locks*) from those that cannot be satisfied right away (*blocked locks*).

All active locks are linked together in the "global file lock list" whose head element is stored in the file_lock_list variable. Similarly, all blocked locks are linked together in the "blocked list" whose head element is stored in the blocked_list variable. The fl_link field is used to insert a lock_file structure in either one of these two lists.

Last but not least, the kernel must keep track of all blocked locks (the "waiters") associated with a given active lock (the "blocker"): this is the purpose of a list that links together all waiters with respect to a given blocker. The fl_block field of the blocker is the dummy head of the list, while the fl_block fields of the waiters store the pointers to the adjacent elements in the list.

FL_FLOCK Locks

An FL_FLOCK lock is always associated with a file object and is thus owned by the process that opened the file (or by all clone processes sharing the same opened file). When a lock is requested and granted, the kernel replaces every other lock that the process is holding on the same file object with the new lock. This happens only when a process wants to change an already owned read lock into a write one, or vice versa. Moreover, when a file object is being freed by the fput( ) function, all FL_FLOCK locks that refer to the file object are destroyed. However, there could be other FL_FLOCK read locks set by other processes for the same file (inode), and they still remain active.

The flock( ) system call allows a process to apply or remove an advisory lock on an open file. It acts on two parameters: the fd file descriptor of the file to be acted upon and a cmd parameter that specifies the lock operation. A cmd parameter of LOCK_SH requires a shared lock for reading, LOCK_EX requires an exclusive lock for writing, and LOCK_UN releases the lock.*

Usually this system call blocks the current process if the request cannot be immediately satisfied, for instance if the process requires an exclusive lock while some other process has already acquired the same lock. However, if the LOCK_NB flag is passed together with the LOCK_SH or LOCK_EX operation, the system call does not block; in other words, if the lock cannot be immediately obtained, the system call returns an error code.

---

* Actually, the flock( ) system call can also establish share-mode mandatory locks by specifying the command LOCK_MAND. However, we'll not further discuss this case.

When the sys_flock( ) service routine is invoked, it performs the following steps:

1. Checks whether fd is a valid file descriptor; if not, returns an error code. Gets the address filp of the corresponding file object.

2. Checks that the process has read and/or write permission on the open file; if not, returns an error code.

3. Gets a new file_lock object lock and initializes it in the appropriate way: the fl_type field is set according to the value of the parameter cmd, the fl_file field is set to the address filp of the file object, the fl_flags field is set to FL_FLOCK, the fl_pid field is set to current->tgid, and the fl_end field is set to -1 to denote the fact that locking refers to the whole file (and not to a portion of it).

4. If the cmd parameter does not include the LOCK_NB bit, it adds to the fl_flags field the FL_SLEEP flag.

5. If the file has a flock file operation, the routine invokes it, passing as its parameters the file object pointer filp, a flag (F_SETLKW or F_SETLK depending on the value of the LOCK_NB bit), and the address of the new file_lock object lock.

6. Otherwise, if the flock file operation is not defined (the common case), invokes flock_lock_file_wait( ) to try to perform the required lock operation. Two parameters are passed: filp, a file object pointer, and lock, the address of the new file_lock object created in step 3.

7. If the file_lock descriptor has not been inserted in the active or blocked lists in the previous step, the routine releases it.

8. Returns 0 in case of success.

The flock_lock_file_wait( ) function executes a cycle consisting of the following steps:

1. Invokes flock_lock_file( ) passing as parameters the file object pointer filp and the address of the new file_lock object lock. This function performs, in turn, the following operations:

   a. Searches the list that filp->f_dentry->d_inode->i_flock points to. If an FL_FLOCK lock for the same file object is found, checks its type (LOCK_SH or LOCK_EX): if it is equal to the type of the new lock, returns 0 (nothing has to be done). Otherwise, the function removes the old element from the list of locks on the inode and the global file lock list, wakes up all processes sleeping in the wait queues of the locks in the fl_block list, and frees the file_lock structure.

   b. If the process is performing an unlock (LOCK_UN), nothing else needs to be done: the lock was nonexisting or it has already been released, thus returns 0.

   c. If an FL_FLOCK lock for the same file object has been found—thus the process is changing an already owned read lock into a write one (or vice versa)—gives some other higher-priority process, in particular every process previously blocked on the old file lock, a chance to run by invoking cond_resched( ).

d. Searches the list of locks on the inode again to verify that no existing FL_ FLOCK lock conflicts with the requested one. There must be no FL_FLOCK write lock in the list, and moreover, there must be no FL_FLOCK lock at all if the process is requesting a write lock.

e. If no conflicting lock exists, it inserts the new file_lock structure into the inode's lock list and into the global file lock list, then returns 0 (success).

f. A conflicting lock has been found: if the FL_SLEEP flag in the fl_flags field is set, it inserts the new lock (the waiter lock) in the circular list of the blocker lock and in the global blocked list.

g. Returns the error code -EAGAIN.

2. Checks the return code of flock_lock_file( ):

a. If the return code is 0 (no conflicting looks), it returns 0 (success).

b. There are incompatibilities. If the FL_SLEEP flag in the fl_flags field is cleared, it releases the lock file_lock descriptor and returns -EAGAIN.

c. Otherwise, there are incompatibilities but the process can sleep: invokes wait_event_interruptible( ) to insert the current process in the lock->fl_ wait wait queue and to suspend it. When the process is awakened (right after the blocker lock has been released), it jumps to step 1 to retry the operation.

FL_POSIX Locks

An FL_POSIX lock is always associated with a process *and* with an inode; the lock is automatically released either when the process dies or when a file descriptor is closed (even if the process opened the same file twice or duplicated a file descriptor). Moreover, FL_POSIX locks are never inherited by a child across a fork( ).

When used to lock files, the fcntl( ) system call acts on three parameters: the fd file descriptor of the file to be acted upon, a cmd parameter that specifies the lock operation, and an fl pointer to a flock data structure[*] stored in the User Mode process address space; its fields are described in Table 12-20.

Table 12-20. The fields of the flock data structure

Type	Field	Description
short	l_type	F_RDLOCK (requests a shared lock), F_WRLOCK (requests an exclusive lock), F_UNLOCK (releases the lock)
short	l_whence	SEEK_SET (from beginning of file), SEEK_CURRENT (from current file pointer), SEEK_END (from end of file)

---

[*] Linux also defines a flock64 structure, which uses 64-bit long integers for the offset and length fields. In the following, we focus on the flock data structure, but the description is valid for flock64 too.

Table 12-20. The fields of the flock data structure (continued)

Type	Field	Description
off_t	l_start	Initial offset of the locked region relative to the value of l_whence
off_t	l_len	Length of locked region (0 means that the region includes all potential writes past the current end of the file)
pid_t	l_pid	PID of the owner

The sys_fcntl( ) service routine behaves differently, depending on the value of the flag set in the cmd parameter:

F_GETLK

    Determines whether the lock described by the flock structure conflicts with some FL_POSIX lock already obtained by another process. In this case, the flock structure is overwritten with the information about the existing lock.

F_SETLK

    Sets the lock described by the flock structure. If the lock cannot be acquired, the system call returns an error code.

F_SETLKW

    Sets the lock described by the flock structure. If the lock cannot be acquired, the system call blocks; that is, the calling process is put to sleep until the lock is available.

F_GETLK64, F_SETLK64, F_SETLKW64

    Identical to the previous ones, but the flock64 data structure is used rather than flock.

The sys_fcntl( ) service routine gets first a file object corresponding to the fd parameter and invokes then fcntl_getlk( ) or fcntl_setlk( ), depending on the command passed as its parameter (F_GETBLK for the former function, F_SETLK or F_SETLKW for the latter one). We'll consider the second case only.

The fcntl_setlk( ) function acts on three parameters: a filp pointer to the file object, a cmd command (F_SETLK or F_SETLKW), and a pointer to a flock data structure. The steps performed are the following:

1. Reads the structure pointed to by the fl parameter in a local variable of type flock.

2. Checks whether the lock should be a mandatory one and the file has a shared memory mapping (see the section "Memory Mapping" in Chapter 16). In this case, the function refuses to create the lock and returns the -EAGAIN error code, because the file is already being accessed by another process.

3. Initializes a new file_lock structure according to the contents of the user's flock structure and to the file size stored in the file's inode.

4. If the command is F_SETLKW, the function sets the FL_SLEEP flag in the fl_flags field of the file_lock structure.

5. If the l_type field in the flock structure is equal to F_RDLCK, it checks whether the process is allowed to read from the file; similarly, if l_type is equal to F_WRLCK, checks whether the process is allowed to write into the file. If not, it returns an error code.

6. Invokes the lock method of the file operations, if defined. Usually for disk-based filesystems, this method is not defined.

7. Invokes __posix_lock_file() passing as parameters the address of the file's inode object and the address of the file_lock object. This function performs, in turn, the following operations:

   a. Invokes posix_locks_conflict() for each FL_POSIX lock in the inode's lock list. The function checks whether the lock conflicts with the requested one. Essentially, there must be no FL_POSIX write lock for the same region in the inode list, and there may be no FL_POSIX lock at all for the same region if the process is requesting a write lock. However, locks owned by the same process never conflict; this allows a process to change the characteristics of a lock it already owns.

   b. If a conflicting lock is found, the function checks whether fcntl() was invoked with the F_SETLKW command. If so, the current process must be suspended: invokes posix_locks_deadlock() to check that no deadlock condition is being created among processes waiting for FL_POSIX locks, then inserts the new lock (waiter lock) both in the blocker list of the conflicting lock (blocker lock) and in the blocked list, and finally returns an error code. Otherwise, if fcntl() was invoked with the F_SETLK command, returns an error code.

   c. As soon as the inode's lock list includes no conflicting lock, the function checks all the FL_POSIX locks of the current process that overlap the file region that the current process wants to lock, and combines and splits adjacent areas as required. For example, if the process requested a write lock for a file region that falls inside a read-locked wider region, the previous read lock is split into two parts covering the nonoverlapping areas, while the central region is protected by the new write lock. In case of overlaps, newer locks always replace older ones.

   d. Inserts the new file_lock structure in the global file lock list and in the inode list.

   e. Returns the value 0 (success).

8. Checks the return code of __posix_lock_file():

   a. If the return code is 0 (no conflicting locks), it returns 0 (success).

   b. There are incompatibilities. If the FL_SLEEP flag in the fl_flags field is cleared, it releases the new file_lock descriptor and returns -EAGAIN.

c. Otherwise, if there are incompatibilities but the process can sleep, it invokes `wait_event_interruptible()` to insert the current process in the `lock->fl_wait` wait queue and to suspend it. When the process is awakened (right after the blocker lock has been released), it jumps to step 7 to retry the operation.

I/O Architecture and Device Drivers

The Virtual File System in the last chapter depends on lower-level functions to carry out each read, write, or other operation in a manner suited to each device. The previous chapter included a brief discussion of how operations are handled by different filesystems. In this chapter, we look at how the kernel invokes the operations on actual devices.

In the section "I/O Architecture," we give a brief survey of the 80×86 I/O architecture. In the section "The Device Driver Model," we introduce the Linux device driver model. Next, in the section "Device Files," we show how the VFS associates a special file called "device file" with each different hardware device, so that application programs can use all kinds of devices in the same way. We then introduce in the section "Device Drivers" some common characteristics of device drivers. Finally, in the section "Character Device Drivers," we illustrate the overall organization of character device drivers in Linux. We'll defer the discussion of block device drivers to the next chapters.

Readers interested in developing device drivers on their own may want to refer to Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman's *Linux Device Drivers,* Third Edition (O'Reilly).

I/O Architecture

To make a computer work properly, data paths must be provided that let information flow between CPU(s), RAM, and the score of I/O devices that can be connected to a personal computer. These data paths, which are denoted as the *buses*, act as the primary communication channels inside the computer.

Any computer has a *system bus* that connects most of the internal hardware devices. A typical system bus is the PCI (*Peripheral Component Interconnect*) bus. Several other types of buses, such as ISA, EISA, MCA, SCSI, and USB, are currently in use. Typically, the same computer includes several buses of different types, linked

together by hardware devices called *bridges*. Two high-speed buses are dedicated to the data transfers to and from the memory chips: the *frontside bus* connects the CPUs to the RAM controller, while the *backside bus* connects the CPUs directly to the external hardware cache. The *host bridge* links together the system bus and the frontside bus.

Any I/O device is hosted by one, and only one, bus. The bus type affects the internal design of the I/O device, as well as how the device has to be handled by the kernel. In this section, we discuss the functional characteristics common to all PC architectures, without giving details about a specific bus type.

The data path that connects a CPU to an I/O device is generically called an *I/O bus*. The 80×86 microprocessors use 16 of their address pins to address I/O devices and 8, 16, or 32 of their data pins to transfer data. The I/O bus, in turn, is connected to each I/O device by means of a hierarchy of hardware components including up to three elements: I/O ports, interfaces, and device controllers. Figure 13-1 shows the components of the I/O architecture.
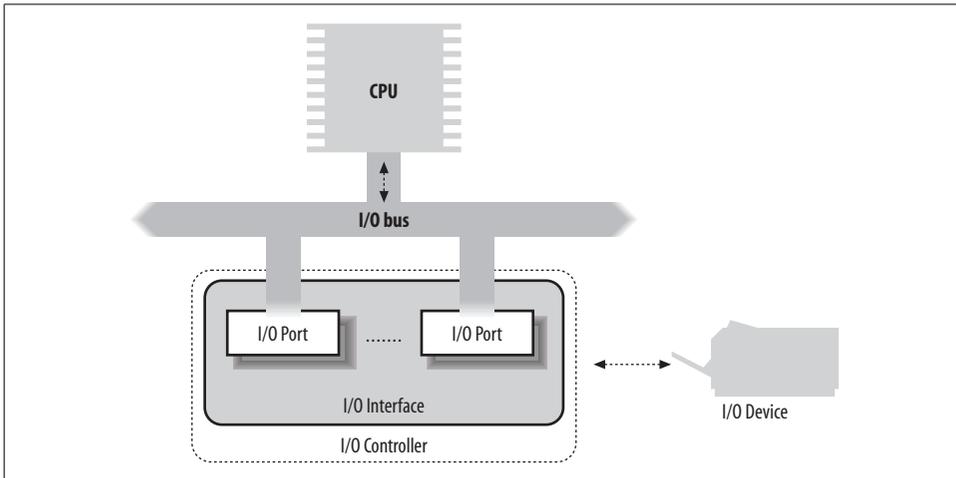
Figure 13-1. PC's I/O architecture

I/O Ports

Each device connected to the I/O bus has its own set of I/O addresses, which are usually called *I/O ports*. In the IBM PC architecture, the *I/O address space* provides up to 65,536 8-bit I/O ports. Two consecutive 8-bit ports may be regarded as a single 16-bit port, which must start on an even address. Similarly, two consecutive 16-bit ports may be regarded as a single 32-bit port, which must start on an address that is a multiple of 4. Four special assembly language instructions called `in`, `ins`, `out`, and `outs` allow the CPU to read from and write into an I/O port. While executing one of these instructions, the CPU selects the required I/O port and transfers the data between a CPU register and the port.

I/O ports may also be mapped into addresses of the physical address space. The processor is then able to communicate with an I/O device by issuing assembly language instructions that operate directly on memory (for instance, `mov`, `and`, `or`, and so on). Modern hardware devices are more suited to mapped I/O, because it is faster and can be combined with DMA.

An important objective for system designers is to offer a unified approach to I/O programming without sacrificing performance. Toward that end, the I/O ports of each device are structured into a set of specialized registers, as shown in Figure 13-2. The CPU writes the commands to be sent to the device into the *device control register* and reads a value that represents the internal state of the device from the *device status register*. The CPU also fetches data from the device by reading bytes from the *device input register* and pushes data to the device by writing bytes into the *device output register*.
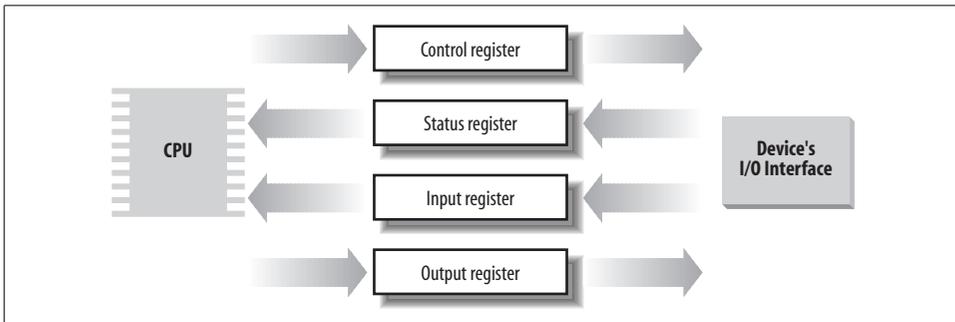
Figure 13-2. Specialized I/O ports

To lower costs, the same I/O port is often used for different purposes. For instance, some bits describe the device state, while others specify the command to be issued to the device. Similarly, the same I/O port may be used as an input register or an output register.

Accessing I/O ports

The `in`, `out`, `ins`, and `outs` assembly language instructions access I/O ports. The following auxiliary functions are included in the kernel to simplify such accesses:

`inb()`, `inw()`, `inl()`
: Read 1, 2, or 4 consecutive bytes, respectively, from an I/O port. The suffix "b," "w," or "l" refers, respectively, to a byte (8 bits), a word (16 bits), and a long (32 bits).

`inb_p()`, `inw_p()`, `inl_p()`
: Read 1, 2, or 4 consecutive bytes, respectively, from an I/O port, and then execute a "dummy" instruction to introduce a pause.

`outb()`, `outw()`, `outl()`
    Write 1, 2, or 4 consecutive bytes, respectively, to an I/O port.

`outb_p()`, `outw_p()`, `outl_p()`
    Write 1, 2, and 4 consecutive bytes, respectively, to an I/O port, and then execute a "dummy" instruction to introduce a pause.

`insb()`, `insw()`, `insl()`
    Read sequences of consecutive bytes in groups of 1, 2, or 4, respectively, from an I/O port. The length of the sequence is specified as a parameter of the functions.

`outsb()`, `outsw()`, `outsl()`
    Write sequences of consecutive bytes, in groups of 1, 2, or 4, respectively, to an I/O port.

While accessing I/O ports is simple, detecting which I/O ports have been assigned to I/O devices may not be easy, in particular, for systems based on an ISA bus. Often a device driver must blindly write into some I/O port to probe the hardware device; if, however, this I/O port is already used by some other hardware device, a system crash could occur. To prevent such situations, the kernel keeps track of I/O ports assigned to each hardware device by means of "resources."

A *resource* represents a portion of some entity that can be exclusively assigned to a device driver. In our case, a resource represents a range of I/O port addresses. The information relative to each resource is stored in a `resource` data structure, whose fields are shown in Table 13-1. All resources of the same kind are inserted in a tree-like data structure; for instance, all resources representing I/O port address ranges are included in a tree rooted at the node `ioport_resource`.

Table 13-1. The fields of the resource data structure

Type	Field	Description
`const char *`	`name`	Description of owner of the resource
`unsigned long`	`start`	Start of the resource range
`unsigned long`	`end`	End of the resource range
`unsigned long`	`flags`	Various flags
`struct resource *`	`parent`	Pointer to parent in the resource tree
`struct resource *`	`sibling`	Pointer to a sibling in the resource tree
`struct resource *`	`child`	Pointer to first child in the resource tree

The children of a node are collected in a list whose first element is pointed to by the `child` field. The `sibling` field points to the next node in the list.

Why use a tree? Well, consider, for instance, the I/O port addresses used by an IDE hard disk interface—let's say from `0xf000` to `0xf00f`. A resource with the `start` field set to `0xf000` and the `end` field set to `0xf00f` is then included in the tree, and the conventional name of the controller is stored in the `name` field. However, the IDE device

driver needs to remember another bit of information, namely that the subrange from 0xf000 to 0xf007 is used for the master disk of the IDE chain, while the subrange from 0xf008 to 0xf00f is used for the slave disk. To do this, the device driver inserts two children below the resource corresponding to the whole range from 0xf000 to 0xf00f, one child for each subrange of I/O ports. As a general rule, each node of the tree must correspond to a subrange of the range associated with the parent. The root of the I/O port resource tree (ioport_resource) spans the whole I/O address space (from port number 0 to 65535).

Each device driver may use the following three functions, passing to them the root node of the resource tree and the address of a resource data structure of interest:

request_resource( )
> Assigns a given range to an I/O device.

allocate_resource( )
> Finds an available range having a given size and alignment in the resource tree; if it exists, assigns the range to an I/O device (mainly used by drivers of PCI devices, which can be configured to use arbitrary port numbers and on-board memory addresses).

release_resource( )
> Releases a given range previously assigned to an I/O device.

The kernel also defines some shortcuts to the above functions that apply to I/O ports: request_region( ) assigns a given interval of I/O ports and release_region( ) releases a previously assigned interval of I/O ports. The tree of all I/O addresses currently assigned to I/O devices can be obtained from the */proc/ioports* file.

I/O Interfaces

An *I/O interface* is a hardware circuit inserted between a group of I/O ports and the corresponding device controller. It acts as an interpreter that translates the values in the I/O ports into commands and data for the device. In the opposite direction, it detects changes in the device state and correspondingly updates the I/O port that plays the role of status register. This circuit can also be connected through an IRQ line to a Programmable Interrupt Controller, so that it issues interrupt requests on behalf of the device.

There are two types of interfaces:

Custom I/O interfaces
> Devoted to one specific hardware device. In some cases, the device controller is located in the same *card** that contains the I/O interface. The devices attached to

---

* Each card must be inserted in one of the available free bus slots of the PC. If the card can be connected to an external device through an external cable, the card supports a suitable connector in the rear panel of the PC.

a custom I/O interface can be either *internal devices* (devices located inside the PC's cabinet) or *external devices* (devices located outside the PC's cabinet).

General-purpose I/O interfaces
Used to connect several different hardware devices. Devices attached to a general-purpose I/O interface are usually external devices.

Custom I/O interfaces

Just to give an idea of how much variety is encompassed by custom I/O interfaces—thus by the devices currently installed in a PC—we'll list some of the most commonly found:

Keyboard interface
Connected to a keyboard controller that includes a dedicated microprocessor. This microprocessor decodes the combination of pressed keys, generates an interrupt, and puts the corresponding scan code in an input register.

Graphic interface
Packed together with the corresponding controller in a graphic card that has its own *frame buffer*, as well as a specialized processor and some code stored in a Read-Only Memory chip (ROM). The frame buffer is an on-board memory containing a description of the current screen contents.

Disk interface
Connected by a cable to the disk controller, which is usually integrated with the disk. For instance, the IDE interface is connected by a 40-wire flat conductor cable to an intelligent disk controller that can be found on the disk itself.

Bus mouse interface
Connected by a cable to the corresponding controller, which is included in the mouse.

Network interface
Packed together with the corresponding controller in a network card used to receive or transmit network packets. Although there are several widely adopted network standards, Ethernet (IEEE 802.3) is the most common.

General-purpose I/O interfaces

Modern PCs include several general-purpose I/O interfaces, which connect a wide range of external devices. The most common interfaces are:

Parallel port
Traditionally used to connect printers, it can also be used to connect removable disks, scanners, backup units, and other computers. The data is transferred 1 byte (8 bits) at a time.

Serial port
Like the parallel port, but the data is transferred 1 bit at a time. It includes a Universal Asynchronous Receiver and Transmitter (UART) chip to string out the

bytes to be sent into a sequence of bits and to reassemble the received bits into bytes. Because it is intrinsically slower than the parallel port, this interface is mainly used to connect external devices that do not operate at a high speed, such as modems, mouses, and printers.

PCMCIA interface

Included mostly on portable computers. The external device, which has the shape of a credit card, can be inserted into and removed from a slot without rebooting the system. The most common PCMCIA devices are hard disks, modems, network cards, and RAM expansions.

SCSI (Small Computer System Interface) interface

A circuit that connects the main PC bus to a secondary bus called the *SCSI bus*. The SCSI-2 bus allows up to eight PCs and external devices—hard disks, scanners, CD-ROM writers, and so on—to be connected. Wide SCSI-2 and the SCSI-3 interfaces allow you to connect 16 devices or more if additional interfaces are present. The *SCSI standard* is the communication protocol used to connect devices via the SCSI bus.

Universal serial bus (USB)

A general-purpose I/O interface that operates at a high speed and may be used for the external devices traditionally connected to the parallel port, the serial port, and the SCSI interface.

Device Controllers

A complex device may require a *device controller* to drive it. Essentially, the controller plays two important roles:

- It interprets the high-level commands received from the I/O interface and forces the device to execute specific actions by sending proper sequences of electrical signals to it.

- It converts and properly interprets the electrical signals received from the device and modifies (through the I/O interface) the value of the status register.

A typical device controller is the *disk controller*, which receives high-level commands such as a "write this block of data" from the microprocessor (through the I/O interface) and converts them into low-level disk operations such as "position the disk head on the right track" and "write the data inside the track." Modern disk controllers are very sophisticated, because they can keep the disk data in on-board fast disk caches and can reorder the CPU high-level requests optimized for the actual disk geometry.

Simpler devices do not have a device controller; examples include the Programmable Interrupt Controller (see the section "Interrupts and Exceptions" in Chapter 4) and the Programmable Interval Timer (see the section "Programmable Interval Timer (PIT)" in Chapter 6).

Several hardware devices include their own memory, which is often called *I/O shared memory*. For instance, all recent graphic cards include tens of megabytes of RAM in the frame buffer, which is used to store the screen image to be displayed on the monitor. We will discuss I/O shared memory in the section "Accessing the I/O Shared Memory" later in this chapter.

The Device Driver Model

Earlier versions of the Linux kernel offered few basic functionalities to the device driver developers: allocating dynamic memory, reserving a range of I/O addresses or an IRQ line, activating an interrupt service routine in response to a device's interrupt. Older hardware devices, in fact, were cumbersome and difficult to program, and two different hardware devices had little in common even if they were hosted on the same bus. Thus, there was no point in trying to offer a unifying model to the device driver developers.

Things are different now. Bus types such as PCI put strong demands on the internal design of the hardware devices; as a consequence, recent hardware devices, even of different classes, support similar functionalities. Drivers for such devices should typically take care of:

- Power management (handling of different voltage levels on the device's power line)
- Plug and play (transparent allocation of resources when configuring the device)
- Hot-plugging (support for insertion and removal of the device while the system is running)

Power management is performed globally by the kernel on every hardware device in the system. For instance, when a battery-powered computer enters the "standby" state, the kernel must force every hardware device (hard disks, graphics card, sound card, network card, bus controllers, and so on) in a low-power state. Thus, each driver of a device that can be put in the "standby" state must include a callback function that puts the hardware device in the low-power state. Moreover, the hardware devices must be put in the "standby" state in a precise order, otherwise some devices could be left in the wrong power state. For instance, the kernel must put in "standby" first the hard disks and then their disk controller, because in the opposite case it would be impossible to send commands to the hard disks.

To implement these kinds of operations, Linux 2.6 provides some data structures and helper functions that offer a unifying view of all buses, devices, and device drivers in the system; this framework is called the *device driver model*.

The sysfs Filesystem

The *sysfs* filesystem is a special filesystem similar to */proc* that is usually mounted on the */sys* directory. The */proc* filesystem was the first special filesystem designed to allow User Mode applications to access kernel internal data structures. The */sysfs* filesystem has essentially the same objective, but it provides additional information on kernel data structures; furthermore, */sysfs* is organized in a more structured way than */proc*. Likely, both */proc* and */sysfs* will continue to coexist in the near future.

A goal of the *sysfs* filesystem is to expose the hierarchical relationships among the components of the device driver model. The related top-level directories of this filesystem are:

block
> The block devices, independently from the bus to which they are connected.

devices
> All hardware devices recognized by the kernel, organized according to the bus in which they are connected.

bus
> The buses in the system, which host the devices.

drivers
> The device drivers registered in the kernel.

class
> The types of devices in the system (audio cards, network cards, graphics cards, and so on); the same class may include devices hosted by different buses and driven by different drivers.

power
> Files to handle the power states of some hardware devices.

firmware
> Files to handle the firmware of some hardware devices.

Relationships between components of the device driver models are expressed in the *sysfs* filesystem as symbolic links between directories and files. For example, the */sys/block/sda/device* file can be a symbolic link to a subdirectory nested in */sys/devices/pci0000:00* representing the SCSI controller connected to the PCI bus. Moreover, the */sys/block/sda/device/block* file is a symbolic link to */sys/block/sda*, stating that this PCI device is the controller of the SCSI disk.

The main role of regular files in the *sysfs* filesystem is to represent attributes of drivers and devices. For instance, the *dev* file in the */sys/block/hda* directory contains the major and minor numbers of the master disk in the first IDE chain.

Kobjects

The core data structure of the device driver model is a generic data structure named *kobject*, which is inherently tied to the *sysfs* filesystem: each kobject corresponds to a directory in that filesystem.

Kobjects are embedded inside larger objects—the so-called "containers"—that describe the components of the device driver model.[*] The descriptors of buses, devices, and drivers are typical examples of containers; for instance, the descriptor of the first partition in the first IDE disk corresponds to the */sys/block/hda/hda1* directory.

Embedding a kobject inside a container allows the kernel to:

- Keep a reference counter for the container
- Maintain hierarchical lists or sets of containers (for instance, a *sysfs* directory associated with a block device includes a different subdirectory for each disk partition)
- Provide a User Mode view for the attributes of the container

Kobjects, ksets, and subsystems

A kobject is represented by a `kobject` data structure, whose fields are listed in Table 13-2.

Table 13-2. The fields of the kobject data structure

Type	Field	Description
`char *`	`k_name`	Pointer to a string holding the name of the container
`char []`	`name`	String holding the name of the container, if it fits in 20 bytes
`struct k_ref`	`kref`	The reference counter for the container
`struct list_head`	`entry`	Pointers for the list in which the kobject is inserted
`struct kobject *`	`parent`	Pointer to the parent kobject, if any
`struct kset *`	`kset`	Pointer to the containing kset
`struct kobj_type *`	`ktype`	Pointer to the kobject type descriptor
`struct dentry *`	`dentry`	Pointer to the dentry of the *sysfs* file associated with the kobject

The `ktype` field points to a `kobj_type` object representing the "type" of the kobject—essentially, the type of the container that includes the kobject. The `kobj_type` data structure includes three fields: a `release` method (executed when the kobject is being freed), a `sysfs_ops` pointer to a table of *sysfs* operations, and a list of default attributes for the *sysfs* filesystem.

---

[*] Kobjects are mainly used to implement the device driver model; however, there is an ongoing effort to change some other kernel components—such as the module subsystem—so as to use them.

The kref field is a structure of type k_ref consisting of a single refcount field. As the name implies, this field is the reference counter for the kobject, but it may act also as the reference counter for the container of the kobject. The kobject_get() and kobject_put() functions increase and decrease, respectively, the reference counter; if the counter reaches the value zero, the resources used by the kobject are released and the release method of the kobj_type object of the kobject is executed. This method, which is usually defined only if the container of the kobject was allocated dynamically, frees the container itself.

The kobjects can be organized in a hierarchical tree by means of *ksets*. A kset is a collection of kobjects of the same type—that is, included in the same type of container. The fields of the kset data structure are listed in Table 13-3.

Table 13-3. The fields of the kset data structure

Type	Field	Description
struct subsystem *	subsys	Pointer to the subsystem descriptor
struct kobj_type *	ktype	Pointer to the kobject type descriptor of the kset
struct list_head	list	Head of the list of kobjects included in the kset
struct kobject	kobj	Embedded kobject (see text)
struct kset_hotplug_ops *	hotplug_ops	Pointer to a table of callback functions for kobject filtering and hot-plugging

The list field is the head of the doubly linked circular list of kobjects included in the kset; the ktype field points to the same kobj_type descriptor shared by all kobjects in the kset.

The kobj field is a kobject embedded in the kset data structure; the parent field of the kobjects contained in the kset points to this embedded kobject. Thus, a kset is a collection of kobjects, but it relies on a kobject of higher level for reference counting and linking in the hierarchical tree. This design choice is code-efficient and allows the greatest flexibility. For instance, the kset_get() and kset_put() functions, which increase and decrease respectively the reference counter of the kset, simply invoke kobject_get() and kobject_put() on the embedded kobject; because the reference counter of a kset is merely the reference counter of the kobj kobject embedded in the kset. Moreover, thanks to the embedded kobject, the kset data structure can be embedded in a "container" object, exactly as for the kobject data structure. Finally, a kset can be made a member of another kset: it suffices to insert the embedded kobject in the higher-level kset.

Collections of ksets called *subsystems* also exist. A subsystem may include ksets of different types, and it is represented by a subsystem data structure having just two fields:

kset
> An embedded kset that stores the ksets included in the subsystem

```
rwsem
```
A read-write semaphore that protects all ksets and kobjects recursively included in the subsystem

Even the `subsystem` data structure can be embedded in a larger "container" object; the reference counter of the container is thus the reference counter of the embedded subsystem—that is, the reference counter of the kobject embedded in the kset embedded in the subsystem. The `subsys_get()` and `subsys_put()` functions respectively increase and decrease this reference counter.

Figure 13-3 illustrates an example of the device driver model hierarchy. The *bus* subsystem includes a *pci* subsystem, which, in turn, includes a *drivers* kset. This kset contains a *serial* kobject—corresponding to the device driver for the serial port—having a single *new-id* attribute.
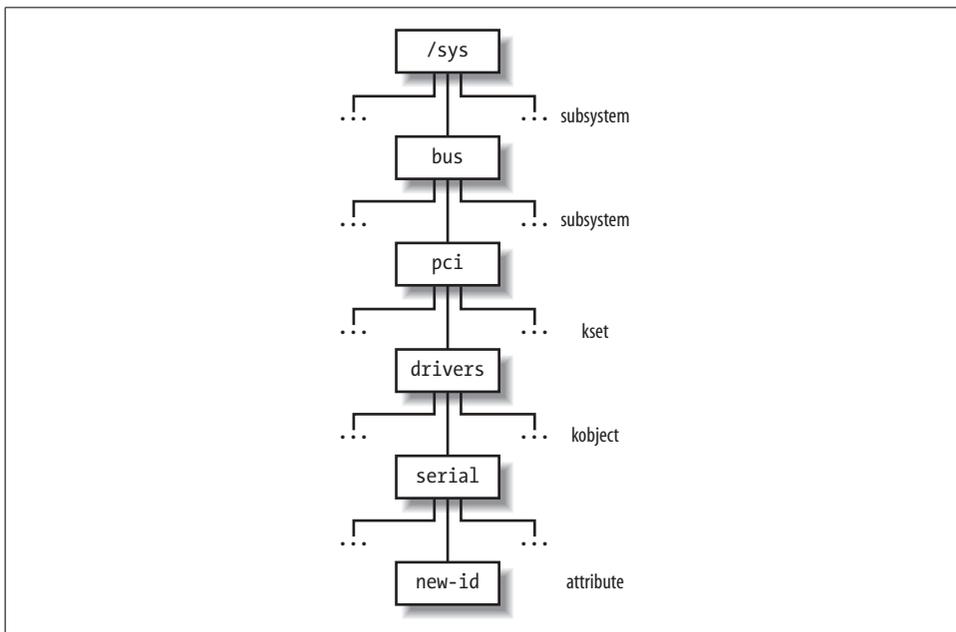
Figure 13-3. An example of device driver model hierarchy

Registering kobjects, ksets, and subsystems

As a general rule, if you want a kobject, kset, or subsystem to appear in the *sysfs* subtree, you must first *register* it. The directory associated with a kobject always appears in the directory of the parent kobject. For instance, the directories of kobjects included in the same kset appear in the directory of the kset itself. Therefore, the structure of the *sysfs* subtree represents the hierarchical relationships between the various registered kobjects and, consequently, between the various container objects.

Usually, the top-level directories of the *sysfs* filesystem are associated with the registered subsystems.

The `kobject_register()` function initializes a kobject and adds the corresponding directory to the *sysfs* filesystem. Before invoking it, the caller should set the `kset` field in the kobject so that it points to the parent kset, if any. The `kobject_unregister()` function removes a kobject's directory from the *sysfs* filesystem. To make life easier for kernel developers, Linux also offers the `kset_register()` and `kset_unregister()` functions, and the `subsystem_register()` and `subsystem_unregister()` functions, but they are essentially wrapper functions around `kobject_register()` and `kobject_unregister()`.

As stated before, many kobject directories include regular files called *attributes*. The `sysfs_create_file()` function receives as its parameters the addresses of a kobject and an attribute descriptor, and creates the special file in the proper directory. Other relationships between the objects represented in the *sysfs* filesystem are established by means of symbolic links: the `sysfs_create_link()` function creates a symbolic link for a given kobject in a directory associated with another kobject.

Components of the Device Driver Model

The device driver model is built upon a handful of basic data structures, which represent buses, devices, device drivers, etc. Let us examine them.

Devices

Each device in the device driver model is represented by a `device` object, whose fields are shown in Table 13-4.

Table 13-4. The fields of the device object

Type	Field	Description
`struct list_head`	`node`	Pointers for the list of sibling devices
`struct list_head`	`bus_list`	Pointers for the list of devices on the same bus type
`struct list_head`	`driver_list`	Pointers for the driver's list of devices
`struct list_head`	`children`	Head of the list of children devices
`struct device *`	`parent`	Pointer to the parent device
`struct kobject`	`kobj`	Embedded kobject
`char []`	`bus_id`	Device position on the hosting bus
`struct bus_type *`	`bus`	Pointer to the hosting bus
`struct device_driver *`	`driver`	Pointer to the controlling device driver
`void *`	`driver_data`	Pointer to private data for the driver
`void *`	`platform_data`	Pointer to private data for legacy device drivers

Table 13-4. The fields of the device object (continued)

Type	Field	Description
struct dev_pm_info	power	Power management information
unsigned long	detach_state	Power state to be entered when unloading the device driver
unsigned long long *	dma_mask	Pointer to the DMA mask of the device (see the later section "Direct Memory Access (DMA)")
unsigned long long	coherent_dma_mask	Mask for coherent DMA of the device
struct list_head	dma_pools	Head of a list of aggregate DMA buffers
struct dma_coherent_mem *	dma_mem	Pointer to a descriptor of the coherent DMA memory used by the device (see the later section "Direct Memory Access (DMA)")
void (*)(struct device *)	release	Callback function for releasing the device descriptor

The device objects are globally collected in the devices_subsys subsystem, which is associated with the */sys/devices* directory (see the earlier section "Kobjects"). The devices are organized hierarchically: a device is the "parent" of some "children" devices if the children devices cannot work properly without the parent device. For instance, in a PCI-based computer, a bridge between the PCI bus and the USB bus is the parent device of every device hosted on the USB bus. The parent field of the device object points to the descriptor of the parent device, the children field is the head of the list of children devices, and the node field stores the pointers to the adjacent elements in the children list. The parenthood relationships between the kobjects embedded in the device objects reflect also the device hierarchy; thus, the structure of the directories below */sys/devices* matches the physical organization of the hardware devices.

Each driver keeps a list of device objects including all managed devices; the driver_list field of the device object stores the pointers to the adjacent elements, while the driver field points to the descriptor of the device driver. For each bus type, moreover, there is a list including all devices that are hosted on the buses of the given type; the bus_list field of the device object stores the pointers to the adjacent elements, while the bus field points to the bus type descriptor.

A reference counter keeps track of the usage of the device object; it is included in the kobj kobject embedded in the descriptor. The counter is increased by invoking get_device(), and it is decreased by invoking put_device().

The device_register() function inserts a new device object in the device driver model, and automatically creates a new directory for it under */sys/devices*. Conversely, the device_unregister() function removes a device from the device driver model.

Usually, the device object is statically embedded in a larger descriptor. For instance, PCI devices are described by pci_dev data structures; the dev field of this structure is

a device object, while the other fields are specific to the PCI bus. The device_register() and device_unregister() functions are executed when the device is being registered or de-registered in the PCI kernel layer.

Drivers

Each driver in the device driver model is described by a device_driver object, whose fields are listed in Table 13-5.

Table 13-5. The fields of the device_driver object

Type	Field	Description
char *	name	Name of the device driver
struct bus_type *	bus	Pointer to descriptor of the bus that hosts the supported devices
struct semaphore	unload_sem	Semaphore to forbid device driver unloading; it is released when the reference counter reaches zero
struct kobject	kobj	Embedded kobject
struct list_head	devices	Head of the list including all devices supported by the driver
struct module *	owner	Identifies the module that implements the device driver, if any (see Appendix B)
int (*)(struct device *)	probe	Method for probing a device (checking that it can be handled by the device driver)
int (*)(struct device *)	remove	Method invoked on a device when it is removed
void (*)(struct device *)	shutdown	Method invoked on a device when it is powered off (shut down)
int (*)(struct device *, unsigned long, unsigned long)	suspend	Method invoked on a device when it is put in low-power state
int (*)(struct device *, unsigned long)	resume	Method invoked on a device when it is put back in the normal state (full power)

The device_driver object includes four methods for handling hot-plugging, plug and play, and power management. The probe method is invoked whenever a bus device driver discovers a device that could possibly be handled by the driver; the corresponding function should probe the hardware to perform further checks on the device. The remove method is invoked on a hot-pluggable device whenever it is removed; it is also invoked on every device handled by the driver when the driver itself is unloaded. The shutdown, suspend, and resume methods are invoked on a device when the kernel must change its power state.

The reference counter included in the kobj kobject embedded in the descriptor keeps track of the usage of the device_driver object. The counter is increased by invoking get_driver(), and it is decreased by invoking put_driver().

The `driver_register()` function inserts a new `device_driver` object in the device driver model, and automatically creates a new directory for it in the *sysfs* filesystem. Conversely, the `driver_unregister()` function removes a driver from the device driver model.

Usually, the `device_driver` object is statically embedded in a larger descriptor. For instance, PCI device drivers are described by `pci_driver` data structures; the `driver` field of this structure is a `device_driver` object, while the other fields are specific to the PCI bus.

Buses

Each bus type supported by the kernel is described by a `bus_type` object, whose fields are listed in Table 13-6.

Table 13-6. The fields of the bus_type object

Type	Field	Description
`char *`	`name`	Name of the bus type
`struct subsystem`	`subsys`	Kobject subsystem associated with this bus type
`struct kset`	`drivers`	The set of kobjects of the drivers
`struct kset`	`devices`	The set of kobjects of the devices
`struct bus_attribute *`	`bus_attrs`	Pointer to the object including the bus attributes and the methods for exporting them to the *sysfs* filesystem
`struct device_attribute *`	`dev_attrs`	Pointer to the object including the device attributes and the methods for exporting them to the *sysfs* filesystem
`struct driver_attribute *`	`drv_attrs`	Pointer to the object including the device driver attributes and the methods for exporting them to the *sysfs* filesystem
`int (*)(struct device *, struct device_driver *)`	`match`	Method for checking whether a given driver supports a given device
`int (*)(struct device *, char **, int, char *, int)`	`hotplug`	Method invoked when a device is being registered
`int (*)(struct device *, unsigned long)`	`suspend`	Method for saving the hardware context state and changing the power level of a device
`int (*)(struct device *)`	`resume`	Method for changing the power level and restoring the hardware context of a device

Each `bus_type` object includes an embedded subsystem; the subsystem stored in the `bus_subsys` variable collects all subsystems embedded in the `bus_type` objects. The `bus_subsys` subsystem is associated with the */sys/bus* directory; thus, for example, there exists a */sys/bus/pci* directory associated with the PCI bus type. The per-bus subsystem typically includes only two ksets named *drivers* and *devices* (corresponding to the drivers and devices fields of the `bus_type` object, respectively).

The `drivers` kset contains the `device_driver` descriptors of all device drivers pertaining to the bus type, while the `devices` kset contains the `device` descriptors of all devices of the given bus type. Because the directories of the devices' kobjects already appear in the *sysfs* filesystem under */sys/devices*, the devices directory of the per-bus subsystem stores symbolic links pointing to directories under */sys/devices*. The `bus_for_each_drv()` and `bus_for_each_dev()` functions iterate over the elements of the lists of drivers and devices, respectively.

The `match` method is executed when the kernel must check whether a given device can be handled by a given driver. Even if each device's identifier has a format specific to the bus that hosts the device, the function that implements the method is usually simple, because it searches the device's identifier in the driver's table of supported identifiers. The `hotplug` method is executed when a device is being registered in the device driver model; the implementing function should add bus-specific information to be passed as environment variables to a User Mode program that is notified about the new available device (see the later section "Device Driver Registration"). Finally, the `suspend` and `resume` methods are executed when a device on a bus of the given type must change its power state.

Classes

Each class is described by a `class` object. All class objects belong to the `class_subsys` subsystem associated with the */sys/class* directory. Each `class` object, moreover, includes an embedded subsystem; thus, for example, there exists a */sys/class/input* directory associated with the *input* class of the device driver model.

Each class object includes a list of `class_device` descriptors, each of which represents a single *logical device* belonging to the class. The `class_device` structure includes a `dev` field that points to a `device` descriptor, thus a logical device always refers to a given device in the device driver model. However, there can be several `class_device` descriptors that refer to the same device. In fact, a hardware device might include several different sub-devices, each of which requires a different User Mode interface. For example, the sound card is a hardware device that usually includes a DSP, a mixer, a game port interface, and so on; each sub-device requires its own User Mode interface, thus it is associated with its own directory in the *sysfs* filesystem.

Device drivers in the same class are expected to offer the same functionalities to the User Mode applications; for instance, all device drivers of sound cards should offer a way to write sound samples to the DSP.

The classes of the device driver model are essentially aimed to provide a standard method for exporting to User Mode applications the interfaces of the logical devices. Each `class_device` descriptor embeds a kobject having an attribute (special file) named *dev*. Such attribute stores the major and minor numbers of the device file that is needed to access to the corresponding logical device (see the next section).

Device Files

As mentioned in Chapter 1, Unix-like operating systems are based on the notion of a file, which is just an information container structured as a sequence of bytes. According to this approach, I/O devices are treated as special files called *device files*; thus, the same system calls used to interact with regular files on disk can be used to directly interact with I/O devices. For example, the same write( ) system call may be used to write data into a regular file or to send it to a printer by writing to the */dev/lp0* device file.

According to the characteristics of the underlying device drivers, device files can be of two types: *block* or *character*. The difference between the two classes of hardware devices is not so clear-cut. At least we can assume the following:

- The data of a block device can be addressed randomly, and the time needed to transfer a data block is small and roughly the same, at least from the point of view of the human user. Typical examples of block devices are hard disks, floppy disks, CD-ROM drives, and DVD players.
- The data of a character device either cannot be addressed randomly (consider, for instance, a sound card), or they can be addressed randomly, but the time required to access a random datum largely depends on its position inside the device (consider, for instance, a magnetic tape driver).

Network cards are a notable exception to this schema, because they are hardware devices that are not directly associated with device files.

Device files have been in use since the early versions of the Unix operating system. A device file is usually a real file stored in a filesystem. Its inode, however, doesn't need to include pointers to blocks of data on the disk (the file's data) because there are none. Instead, the inode must include an identifier of the hardware device corresponding to the character or block device file.

Traditionally, this identifier consists of the type of device file (character or block) and a pair of numbers. The first number, called the *major number*, identifies the device type. Traditionally, all device files that have the same major number and the same type share the same set of file operations, because they are handled by the same device driver. The second number, called the *minor number*, identifies a specific device among a group of devices that share the same major number. For instance, a group of disks managed by the same disk controller have the same major number and different minor numbers.

The mknod( ) system call is used to create device files. It receives the name of the device file, its type, and the major and minor numbers as its parameters. Device files are usually included in the */dev* directory. Table 13-7 illustrates the attributes of some device files. Notice that character and block devices have independent numbering, so block device (3,0) is different from character device (3,0).

Table 13-7. Examples of device files

Name	Type	Major	Minor	Description
/dev/fd0	block	2	0	Floppy disk
/dev/hda	block	3	0	First IDE disk
/dev/hda2	block	3	2	Second primary partition of first IDE disk
/dev/hdb	block	3	64	Second IDE disk
/dev/hdb3	block	3	67	Third primary partition of second IDE disk
/dev/ttyp0	char	3	0	Terminal
/dev/console	char	5	1	Console
/dev/lp1	char	6	1	Parallel printer
/dev/ttyS0	char	4	64	First serial port
/dev/rtc	char	10	135	Real-time clock
/dev/null	char	1	3	Null device (black hole)

Usually, a device file is associated with a hardware device (such as a hard disk—for instance, */dev/hda*) or with some physical or logical portion of a hardware device (such as a disk partition—for instance, */dev/hda2*). In some cases, however, a device file is not associated with any real hardware device, but represents a fictitious logical device. For instance, */dev/null* is a device file corresponding to a "black hole;" all data written into it is simply discarded, and the file always appears empty.

As far as the kernel is concerned, the name of the device file is irrelevant. If you create a device file named */tmp/disk* of type "block" with the major number 3 and minor number 0, it would be equivalent to the */dev/hda* device file shown in the table. On the other hand, device filenames may be significant for some application programs. For example, a communication program might assume that the first serial port is associated with the */dev/ttyS0* device file. But most application programs can be configured to interact with arbitrarily named device files.

User Mode Handling of Device Files

In traditional Unix systems (and in earlier versions of Linux), the major and minor numbers of the device files are 8 bits long. Thus, there could be at most 65,536 block device files and 65,536 character device files. You might expect they will suffice, but unfortunately they don't.

The real problem is that device files are traditionally allocated once and forever in the */dev* directory; therefore, each logical device in the system should have an associated device file with a well-defined device number. The official registry of allocated device numbers and */dev* directory nodes is stored in the *Documentation/devices.txt* file; the macros corresponding to the major numbers of the devices may also be found in the *include/linux/major.h* file.

Unfortunately, the number of different hardware devices is so large nowadays that almost all device numbers have already been allocated. The official registry of device numbers works well for the average Linux system; however, it may not be well suited for large-scale systems. Furthermore, high-end systems may use hundreds or thousands of disks of the same type, and an 8-bit minor number is not sufficient. For instance, the registry reserves device numbers for 16 SCSI disks having 15 partitions each; if a high-end system has more than 16 SCSI disks, the standard assignment of major and minor numbers has to be changed—a non trivial task that requires modifying the kernel source code and makes the system hard to maintain.

In order to solve this kind of problem, the size of the device numbers has been increased in Linux 2.6: the major number is now encoded in 12 bits, while the minor number is encoded in 20 bits. Both numbers are usually kept in a single 32-bit variable of type dev_t; the MAJOR and MINOR macros extract the major and minor numbers, respectively, from a dev_t value, while the MKDEV macro encodes the two device numbers in a dev_t value. For backward compatibility, the kernel handles properly old device files encoded with 16-bit device numbers.

The additional available device numbers are not being statically allocated in the official registry, because they should be used only when dealing with unusual demands for device numbers. Actually, today's preferred way to deal with device files is highly dynamic, both in the device number assignment and in the device file creation.

Dynamic device number assignment

Each device driver specifies in the registration phase the range of device numbers that it is going to handle (see the later section "Device Driver Registration"). The driver can, however, require the allocation of an interval of device numbers without specifying the exact values: in this case, the kernel allocates a suitable range of numbers and assigns them to the driver.

Therefore, device drivers of new hardware devices no longer require an assignment in the official registry of device numbers; they can simply use whatever numbers are currently available in the system.

In this case, however, the device file cannot be created once and forever; it must be created right after the device driver initialization with the proper major and minor numbers. Thus, there must be a standard way to export the device numbers used by each driver to the User Mode applications. As we have seen in the earlier section "Components of the Device Driver Model," the device driver model provides an elegant solution: the major and minor numbers are stored in the *dev* attributes contained in the subdirectories of */sys/class*.

Dynamic device file creation

The Linux kernel can create the device files dynamically: there is no need to fill the */dev* directory with the device files of every conceivable hardware device, because the device

files can be created "on demand." Thanks to the device driver model, the kernel 2.6 offers a very simple way to do so. A set of User Mode programs, collectively known as the *udev* toolset, must be installed in the system. At the system startup the */dev* directory is emptied, then a *udev* program scans the subdirectories of */sys/class* looking for the *dev* files. For each such file, which represents a combination of major and minor number for a logical device supported by the kernel, the program creates a corresponding device file in */dev*. It also assigns device filenames and creates symbolic links according to a configuration file, in such a way to resemble the traditional naming scheme for Unix device files. Eventually, */dev* is filled with the device files of all devices supported by the kernel on this system, and nothing else.

Often a device file is created after the system has been initialized. This happens either when a module containing a device driver for a still unsupported device is loaded, or when a hot-pluggable device—such as a USB peripheral—is plugged in the system. The *udev* toolset can automatically create the corresponding device file, because the device driver model supports *device hotplugging*. Whenever a new device is discovered, the kernel spawns a new process that executes the User Mode */sbin/hotplug* shell script,[*] passing to it any useful information on the discovered device as environment variables. The User Mode scripts usually reads a configuration file and takes care of any operation required to complete the initialization of the new device. If *udev* is installed, the script also creates the proper device file in the */dev* directory.

VFS Handling of Device Files

Device files live in the system directory tree but are intrinsically different from regular files and directories. When a process accesses a regular file, it is accessing some data blocks in a disk partition through a filesystem; when a process accesses a device file, it is just driving a hardware device. For instance, a process might access a device file to read the room temperature from a digital thermometer connected to the computer. It is the VFS's responsibility to hide the differences between device files and regular files from application programs.

To do this, the VFS changes the default file operations of a device file when it is opened; as a result, each system call on the device file is translated to an invocation of a device-related function instead of the corresponding function of the hosting filesystem. The device-related function acts on the hardware device to perform the operation requested by the process.[†]

Let's suppose that a process executes an open( ) system call on a device file (either of type block or character). The operations performed by the system call have already

---

[*] The pathname of the User Mode program invoked on hot-plugging events can be changed by writing into the */proc/sys/kernel/hotplug* file.

[†] Notice that, thanks to the name-resolving mechanism explained in the section "Pathname Lookup" in Chapter 12, symbolic links to device files work just like device files.

been described in the section "The open() System Call" in Chapter 12. Essentially, the corresponding service routine resolves the pathname to the device file and sets up the corresponding inode object, dentry object, and file object.

The inode object is initialized by reading the corresponding inode on disk through a suitable function of the filesystem (usually `ext2_read_inode()` or `ext3_read_inode()`; see Chapter 18). When this function determines that the disk inode is relative to a device file, it invokes `init_special_inode()`, which initializes the `i_rdev` field of the inode object to the major and minor numbers of the device file, and sets the `i_fop` field of the inode object to the address of either the `def_blk_fops` or the `def_chr_fops` file operation table, according to the type of device file. The service routine of the open() system call also invokes the `dentry_open()` function, which allocates a new file object and sets its `f_op` field to the address stored in `i_fop`—that is, to the address of `def_blk_fops` or `def_chr_fops` once again. Thanks to these two tables, every system call issued on a device file will activate a device driver's function rather than a function of the underlying filesystem.

Device Drivers

A *device driver* is the set of kernel routines that makes a hardware device respond to the programming interface defined by the canonical set of VFS functions (*open*, *read*, *lseek*, *ioctl*, and so forth) that control a device. The actual implementation of all these functions is delegated to the device driver. Because each device has a different I/O controller, and thus different commands and different state information, most I/O devices have their own drivers.

There are many types of device drivers. They mainly differ in the level of support that they offer to the User Mode applications, as well as in their buffering strategies for the data collected from the hardware devices. Because these choices greatly influence the internal structure of a device driver, we discuss them in the sections "Direct Memory Access (DMA)" and "Buffering Strategies for Character Devices."

A device driver does not consist only of the functions that implement the device file operations. Before using a device driver, several activities must have taken place. We'll examine them in the following sections.

Device Driver Registration

We know that each system call issued on a device file is translated by the kernel into an invocation of a suitable function of a corresponding device driver. To achieve this, a device driver must *register* itself. In other words, registering a device driver means allocating a new `device_driver` descriptor, inserting it in the data structures of the device driver model (see the earlier section "Components of the Device Driver Model"), and linking it to the corresponding device file(s). Accesses to device files

whose corresponding drivers have not been previously registered return the error code -ENODEV.

If a device driver is statically compiled in the kernel, its registration is performed during the kernel initialization phase. Conversely, if a device driver is compiled as a kernel module (see Appendix B), its registration is performed when the module is loaded. In the latter case, the device driver can also unregister itself when the module is unloaded.

Let us consider, for instance, a generic PCI device. To properly handle it, its device driver must allocate a descriptor of type `pci_driver`, which is used by the PCI kernel layer to handle the device. After having initialized some fields of this descriptor, the device driver invokes the `pci_register_driver()` function. Actually, the `pci_driver` descriptor includes an embedded `device_driver` descriptor (see the earlier section "Components of the Device Driver Model"); the `pci_register_function()` simply initializes the fields of the embedded driver descriptor and invokes `driver_register()` to insert the driver in the data structures of the device driver model.

When a device driver is being registered, the kernel looks for unsupported hardware devices that could be possibly handled by the driver. To do this, it relies on the `match` method of the relevant `bus_type` bus type descriptor, and on the `probe` method of the `device_driver` object. If a hardware device that can be handled by the driver is discovered, the kernel allocates a `device` object and invokes `device_register()` to insert the device in the device driver model.

Device Driver Initialization

Registering a device driver and initializing it are two different things. A device driver is registered as soon as possible, so User Mode applications can use it through the corresponding device files. In contrast, a device driver is initialized at the last possible moment. In fact, initializing a driver means allocating precious resources of the system, which are therefore not available to other drivers.

We already have seen an example in the section "I/O Interrupt Handling" in Chapter 4: the assignment of IRQs to devices is usually made dynamically, right before using them, because several devices may share the same IRQ line. Other resources that can be allocated at the last possible moment are page frames for DMA transfer buffers and the DMA channel itself (for old non-PCI devices such as the floppy disk driver).

To make sure the resources are obtained when needed but are not requested in a redundant manner when they have already been granted, device drivers usually adopt the following schema:

- A usage counter keeps track of the number of processes that are currently accessing the device file. The counter is increased in the open method of the device file and decreased in the release method.[*]

- The open method checks the value of the usage counter before the increment. If the counter is zero, the device driver must allocate the resources and enable interrupts and DMA on the hardware device.

- The release method checks the value of the usage counter after the decrement. If the counter is zero, no more processes are using the hardware device. If so, the method disables interrupts and DMA on the I/O controller, and then releases the allocated resources.

Monitoring I/O Operations

The duration of an I/O operation is often unpredictable. It can depend on mechanical considerations (the current position of a disk head with respect to the block to be transferred), on truly random events (when a data packet arrives on the network card), or on human factors (when a user presses a key on the keyboard or when she notices that a paper jam occurred in the printer). In any case, the device driver that started an I/O operation must rely on a monitoring technique that signals either the termination of the I/O operation or a time-out.

In the case of a terminated operation, the device driver reads the status register of the I/O interface to determine whether the I/O operation was carried out successfully. In the case of a time-out, the driver knows that something went wrong, because the maximum time interval allowed to complete the operation elapsed and nothing happened.

The two techniques available to monitor the end of an I/O operation are called the *polling mode* and the *interrupt mode*.

Polling mode

According to this technique, the CPU checks (polls) the device's status register repeatedly until its value signals that the I/O operation has been completed. We have already encountered a technique based on polling in the section "Spin Locks" in Chapter 5: when a processor tries to acquire a busy spin lock, it repeatedly polls the variable until its value becomes 0. However, polling applied to I/O operations is usually more elaborate, because the driver must also remember to check for possible time-outs. A simple example of polling looks like the following:

```
for (;;) {
    if (read_status(device) & DEVICE_END_OPERATION) break;
```

---

[*] More precisely, the usage counter keeps track of the number of file objects referring to the device file, because clone processes could share the same file object.

```
        if (--count == 0) break;
    }
```

The count variable, which was initialized before entering the loop, is decreased at each iteration, and thus can be used to implement a rough time-out mechanism. Alternatively, a more precise time-out mechanism could be implemented by reading the value of the tick counter jiffies at each iteration (see the section "Updating the Time and Date" in Chapter 6) and comparing it with the old value read before starting the wait loop.

If the time required to complete the I/O operation is relatively high, say in the order of milliseconds, this schema becomes inefficient because the CPU wastes precious machine cycles while waiting for the I/O operation to complete. In such cases, it is preferable to voluntarily relinquish the CPU after each polling operation by inserting an invocation of the schedule( ) function inside the loop.

Interrupt mode

Interrupt mode can be used only if the I/O controller is capable of signaling, via an IRQ line, the end of an I/O operation.

We'll show how interrupt mode works on a simple case. Let's suppose we want to implement a driver for a simple input character device. When the user issues a read( ) system call on the corresponding device file, an input command is sent to the device's control register. After an unpredictably long time interval, the device puts a single byte of data in its input register. The device driver then returns this byte as the result of the read( ) system call.

This is a typical case in which it is preferable to implement the driver using the interrupt mode. Essentially, the driver includes two functions:

1. The foo_read( ) function that implements the read method of the file object.
2. The foo_interrupt( ) function that handles the interrupt.

The foo_read( ) function is triggered whenever the user reads the device file:

```
ssize_t foo_read(struct file *filp, char *buf, size_t count,
                 loff_t *ppos)
{
    foo_dev_t * foo_dev = filp->private_data;
    if (down_interruptible(&foo_dev->sem)
        return -ERESTARTSYS;
    foo_dev->intr = 0;
    outb(DEV_FOO_READ, DEV_FOO_CONTROL_PORT);
    wait_event_interruptible(foo_dev->wait, (foo_dev->intr==1));
    if (put_user(foo_dev->data, buf))
        return -EFAULT;
    up(&foo_dev->sem);
    return 1;
}
```

The device driver relies on a custom descriptor of type `foo_dev_t`; it includes a semaphore `sem` that protects the hardware device from concurrent accesses, a wait queue `wait`, a flag `intr` that is set when the device issues an interrupt, and a single-byte buffer `data` that is written by the interrupt handler and read by the read method. In general, all I/O drivers that use interrupts rely on data structures accessed by both the interrupt handler and the `read` and `write` methods. The address of the `foo_dev_t` descriptor is usually stored in the `private_data` field of the device file's file object or in a global variable.

The main operations of the `foo_read( )` function are the following:

1. Acquires the `foo_dev->sem` semaphore, thus ensuring that no other process is accessing the device.

2. Clears the `intr` flag.

3. Issues the read command to the I/O device.

4. Executes `wait_event_interruptible` to suspend the process until the `intr` flag becomes 1. This macro is described in the section "Wait queues" in Chapter 3.

After some time, our device issues an interrupt to signal that the I/O operation is completed and that the data is ready in the proper `DEV_FOO_DATA_PORT` data port. The interrupt handler sets the `intr` flag and wakes the process. When the scheduler decides to reexecute the process, the second part of `foo_read( )` is executed and does the following:

1. Copies the character ready in the `foo_dev->data` variable into the user address space.

2. Terminates after releasing the `foo_dev->sem` semaphore.

For simplicity, we didn't include any time-out control. In general, time-out control is implemented through static or dynamic timers (see Chapter 6); the timer must be set to the right time before starting the I/O operation and removed when the operation terminates.

Let's now look at the code of the `foo_interrupt( )` function:

```
irqreturn_t foo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    foo->data = inb(DEV_FOO_DATA_PORT);
    foo->intr = 1;
    wake_up_interruptible(&foo->wait);
    return 1;
}
```

The interrupt handler reads the character from the input register of the device and stores it in the `data` field of the `foo_dev_t` descriptor of the device driver pointed to by the `foo` global variable. It then sets the `intr` flag and invokes `wake_up_interruptible( )` to wake the process blocked in the `foo->wait` wait queue.

Notice that none of the three parameters are used by our interrupt handler. This is a rather common case.

Accessing the I/O Shared Memory

Depending on the device and on the bus type, I/O shared memory in the PC's architecture may be mapped within different physical address ranges. Typically:

For most devices connected to the ISA bus
> The I/O shared memory is usually mapped into the 16-bit physical addresses ranging from `0xa0000` to `0xfffff`; this gives rise to the "hole" between 640 KB and 1 MB mentioned in the section "Physical Memory Layout" in Chapter 2.

For devices connected to the PCI bus
> The I/O shared memory is mapped into 32-bit physical addresses near the 4 GB boundary. This kind of device is much simpler to handle.

A few years ago, Intel introduced the *Accelerated Graphics Port* (*AGP*) standard, which is an enhancement of PCI for high-performance graphic cards. Beside having its own I/O shared memory, this kind of card is capable of directly addressing portions of the motherboard's RAM by means of a special hardware circuit named *Graphics Address Remapping Table* (*GART*). The GART circuitry enables AGP cards to sustain much higher data transfer rates than older PCI cards. From the kernel's point of view, however, it doesn't really matter where the physical memory is located, and GART-mapped memory is handled like the other kinds of I/O shared memory.

How does a device driver access an I/O shared memory location? Let's start with the PC's architecture, which is relatively simple to handle, and then extend the discussion to other architectures.

Remember that kernel programs act on linear addresses, so the I/O shared memory locations must be expressed as addresses greater than `PAGE_OFFSET`. In the following discussion, we assume that `PAGE_OFFSET` is equal to `0xc0000000`—that is, that the kernel linear addresses are in the fourth gigabyte.

Device drivers must translate I/O physical addresses of I/O shared memory locations into linear addresses in kernel space. In the PC architecture, this can be achieved simply by ORing the 32-bit physical address with the `0xc0000000` constant. For instance, suppose the kernel needs to store the value in the I/O location at physical address `0x000b0fe4` in `t1` and the value in the I/O location at physical address `0xfc000000` in `t2` . One might think that the following statements could do the job:

```
t1 = *((unsigned char *)(0xc00b0fe4));
t2 = *((unsigned char *)(0xfc000000));
```

During the initialization phase, the kernel maps the available RAM's physical addresses into the initial portion of the fourth gigabyte of the linear address space.

Therefore, the Paging Unit maps the `0xc00b0fe4` linear address appearing in the first statement back to the original I/O physical address `0x000b0fe4`, which falls inside the "ISA hole" between 640 KB and 1 MB (see the section "Paging in Linux" in Chapter 2). This works fine.

There is a problem, however, for the second statement, because the I/O physical address is greater than the last physical address of the system RAM. Therefore, the `0xfc000000` linear address does not correspond to the `0xfc000000` physical address. In such cases, the kernel Page Tables must be modified to include a linear address that maps the I/O physical address. This can be done by invoking the `ioremap()` or `ioremap_nocache()` functions. The first function, which is similar to `vmalloc()`, invokes `get_vm_area()` to create a new `vm_struct` descriptor (see the section "Descriptors of Noncontiguous Memory Areas" in Chapter 8) for a linear address interval that has the size of the required I/O shared memory area. The functions then update the corresponding Page Table entries of the canonical kernel Page Tables appropriately. The `ioremap_nocache()` function differs from `ioremap()` in that it also disables the hardware cache when referencing the remapped linear addresses properly.

The correct form for the second statement might therefore look like:

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = *((unsigned char *)(io_mem + 0x100000));
```

The first statement creates a new 2 MB linear address interval, which maps physical addresses starting from `0xfb000000`; the second one reads the memory location that has the `0xfc000000` address. To remove the mapping later, the device driver must use the `iounmap()` function.

On some architectures other than the PC, I/O shared memory cannot be accessed by simply dereferencing the linear address pointing to the physical memory location. Therefore, Linux defines the following architecture-dependent functions, which should be used when accessing I/O shared memory:

`readb()`, `readw()`, `readl()`
    Reads 1, 2, or 4 bytes, respectively, from an I/O shared memory location

`writeb()`, `writew()`, `writel()`
    Writes 1, 2, or 4 bytes, respectively, into an I/O shared memory location

`memcpy_fromio()`, `memcpy_toio()`
    Copies a block of data from an I/O shared memory location to dynamic memory and vice versa

`memset_io()`
    Fills an I/O shared memory area with a fixed value

The recommended way to access the `0xfc000000` I/O location is thus:

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

Thanks to these functions, all dependencies on platform-specific ways of accessing the I/O shared memory can be hidden.

Direct Memory Access (DMA)

In the original PC architecture, the CPU is the only *bus master* of the system, that is, the only hardware device that drives the address/data bus in order to fetch and store values in the RAM's locations. With more modern bus architectures such as PCI, each peripheral can act as bus master, if provided with the proper circuitry. Thus, nowadays all PCs include auxiliary *DMA circuits*, which can transfer data between the RAM and an I/O device. Once activated by the CPU, the DMA is able to continue the data transfer on its own; when the data transfer is completed, the DMA issues an interrupt request. The conflicts that occur when CPUs and DMA circuits need to access the same memory location at the same time are resolved by a hardware circuit called a *memory arbiter* (see the section "Atomic Operations" in Chapter 5).

The DMA is mostly used by disk drivers and other devices that transfer a large number of bytes at once. Because setup time for the DMA is relatively high, it is more efficient to directly use the CPU for the data transfer when the number of bytes is small.

The first DMA circuits for the old ISA buses were complex, hard to program, and limited to the lower 16 MB of physical memory. More recent DMA circuits for the PCI and SCSI buses rely on dedicated hardware circuits in the buses and make life easier for device driver developers.

Synchronous and asynchronous DMA

A device driver can use the DMA in two different ways called *synchronous DMA* and *asynchronous DMA*. In the first case, the data transfers are triggered by processes; in the second case the data transfers are triggered by hardware devices.

An example of synchronous DMA is a sound card that is playing a sound track. A User Mode application writes the sound data (called *samples*) on a device file associated with the *digital signal processor* (*DSP*) of the sound card. The device driver of the sound card accumulates these samples in a kernel buffer. At the same time, the device driver instructs the sound card to copy the samples from the kernel buffer to the DSP with a well-defined timing. When the sound card finishes the data transfer, it raises an interrupt, and the device driver checks whether the kernel buffer still contains samples yet to be played; if so, the driver activates another DMA data transfer.

An example of asynchronous DMA is a network card that is receiving a frame (data packet) from a LAN. The peripheral stores the frame in its I/O shared memory, then raises an interrupt. The device driver of the network card acknowledges the interrupt, then instructs the peripheral to copy the frame from the I/O shared memory into a kernel buffer. When the data transfer completes, the network card raises another interrupt, and the device driver notifies the upper kernel layer about the new frame.

Helper functions for DMA transfers

When designing a driver for a device that makes use of DMA, the developer should write code that is both architecture-independent and, as far as DMA is concerned, bus-independent. This goal is now feasible thanks to the rich set of DMA helper functions provided by the kernel. These helper functions hide the differences in the DMA mechanisms of the various hardware architectures.

There are two subsets of DMA helper functions: an older subset provides architecture-independent functions for PCI devices; a more recent subset ensures both bus and architecture independence. We'll now examine some of these functions while pointing out some hardware peculiarities of DMAs.

Bus addresses

Every DMA transfer involves (at least) one memory buffer, which contains the data to be read or written by the hardware device. In general, before activating the transfer, the device driver must ensure that the DMA circuit can directly access the RAM locations.

Until now we have distinguished three kinds of memory addresses: logical and linear addresses, which are used internally by the CPU, and physical addresses, which are the memory addresses used by the CPU to physically drive the data bus. However, there is a fourth kind of memory address: the so-called *bus address*. It corresponds to the memory addresses used by all hardware devices except the CPU to drive the data bus.

Why should the kernel be concerned at all about bus addresses? Well, in a DMA operation, the data transfer takes place without CPU intervention; the data bus is driven directly by the I/O device and the DMA circuit. Therefore, when the kernel sets up a DMA operation, it must write the bus address of the memory buffer involved in the proper I/O ports of the DMA or I/O device.

In the 80×86 architecture, bus addresses coincide with physical addresses. However, other architectures such as Sun's SPARC and Hewlett-Packard's Alpha include a hardware circuit called the *I/O Memory Management Unit (IO-MMU)*, analog to the paging unit of the microprocessor, which maps physical addresses into bus addresses. All I/O drivers that make use of DMAs must set up properly the IO-MMU before starting the data transfer.

Different buses have different bus address sizes. For instance, bus addresses for ISA are 24-bits long, thus in the 80×86 architecture DMA transfers can be done only on the lower 16 MB of physical memory—that's why the memory for the buffer used by such DMA has to be allocated in the ZONE_DMA memory zone with the GFP_DMA flag. The original PCI standard defines bus addresses of 32 bits; however, some PCI hardware devices have been originally designed for the ISA bus, thus they still cannot access RAM locations above physical address 0x00ffffff. The recent PCI-X standard uses 64-bit bus addresses and allows DMA circuits to address directly the high memory.

In Linux, the `dma_addr_t` type represents a generic bus address. In the 80×86 architecture `dma_addr_t` corresponds to a 32-bit integer, unless the kernel supports PAE (see the section "The Physical Address Extension (PAE) Paging Mechanism" in Chapter 2), in which case `dma_addr_t` corresponds to a 64-bit integer.

The `pci_set_dma_mask()` and `dma_set_mask()` helper functions check whether the bus accepts a given size for the bus addresses (mask) and, if so, notify the bus layer that the given peripheral will use that size for its bus addresses.

Cache coherency

The system architecture does not necessarily offer a coherency protocol between the hardware cache and the DMA circuits at the hardware level, so the DMA helper functions must take into consideration the hardware cache when implementing DMA mapping operations. To see why, suppose that the device driver fills the memory buffer with some data, then immediately instructs the hardware device to read that data with a DMA transfer. If the DMA accesses the physical RAM locations but the corresponding hardware cache lines have not yet been written to RAM, then the hardware device fetches the old values of the memory buffer.

Device driver developers may handle DMA buffers in two different ways by making use of two different classes of helper functions. Using Linux terminology, the developer chooses between two different *DMA mapping types*:

Coherent DMA mapping
> When using this mapping, the kernel ensures that there will be no cache coherency problems between the memory and the hardware device; this means that every write operation performed by the CPU on a RAM location is immediately visible to the hardware device, and vice versa. This type of mapping is also called "synchronous" or "consistent."

Streaming DMA mapping
> When using this mapping, the device driver must take care of cache coherency problems by using the proper synchronization helper functions. This type of mapping is also called "asynchronous" or "non-coherent."

In the 80×86 architecture there are never cache coherency problems when using the DMA, because the hardware devices themselves take care of "snooping" the accesses to the hardware caches. Therefore, a driver for a hardware device designed specifically for the 80×86 architecture may choose either one of the two DMA mapping types: they are essentially equivalent. On the other hand, in many architectures—such as MIPS, SPARC, and some models of PowerPC—hardware devices do not always snoop in the hardware caches, so cache coherency problems arise. In general, choosing the proper DMA mapping type for an architecture-independent driver is not trivial.

As a general rule, if the buffer is accessed in unpredictable ways by the CPU and the DMA processor, coherent DMA mapping is mandatory (for instance, buffers for SCSI adapters' command data structures). In other cases, streaming DMA mapping is preferable, because in some architectures handling the coherent DMA mapping is cumbersome and may lead to lower system performance.

Helper functions for coherent DMA mappings

Usually, the device driver allocates the memory buffer and establishes the coherent DMA mapping in the initialization phase; it releases the mapping and the buffer when it is unloaded. To allocate a memory buffer and to establish a coherent DMA mapping, the kernel provides the architecture-dependent `pci_alloc_consistent()` and `dma_alloc_coherent()` functions. They both return the linear address and the bus address of the new buffer. In the 80×86 architecture, they return the linear address and the physical address of the new buffer. To release the mapping and the buffer, the kernel provides the `pci_free_consistent()` and the `dma_free_coherent()` functions.

Helper functions for streaming DMA mappings

Memory buffers for streaming DMA mappings are usually mapped just before the transfer and unmapped thereafter. It is also possible to keep the same mapping among several DMA transfers, but in this case the device driver developer must be aware of the hardware cache lying between the memory and the peripheral.

To set up a streaming DMA transfer, the driver must first dynamically allocate the memory buffer by means of the zoned page frame allocator (see the section "The Zoned Page Frame Allocator" in Chapter 8) or the generic memory allocator (see the section "General Purpose Objects" in Chapter 8). Then, the drivers must establish the streaming DMA mapping by invoking either the `pci_map_single()` or the `dma_map_single()` function, which receives as its parameter the linear address of the buffer and returns its bus address. To release the mapping, the driver invokes the corresponding `pci_unmap_single()` or `dma_unmap_single()` functions.

To avoid cache coherency problems, right before starting a DMA transfer from the RAM to the device, the driver should invoke `pci_dma_sync_single_for_device()` or `dma_sync_single_for_device()`, which flush, if necessary, the cache lines corresponding to the DMA buffer. Similarly, a device driver should not access a memory buffer right after the end of a DMA transfer from the device to the RAM: instead, before reading the buffer, the driver should invoke `pci_dma_sync_single_for_cpu()` or `dma_sync_single_for_cpu()`, which invalidate, if necessary, the corresponding hardware cache lines. In the 80×86 architecture, these functions do almost nothing, because the coherency between hardware caches and DMAs is maintained by the hardware.

Even buffers in high memory (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8) can be used for DMA transfers; the developer uses `pci_map_page()`—or `dma_map_page()`—passing to it the descriptor address of the page

including the buffer and the offset of the buffer inside the page. Correspondingly, to release the mapping of the high memory buffer, the developer uses `pci_unmap_page()` or `dma_unmap_page()`.

Levels of Kernel Support

The Linux kernel does not fully support all possible existing I/O devices. Generally speaking, in fact, there are three possible kinds of support for a hardware device:

No support at all
> The application program interacts directly with the device's I/O ports by issuing suitable `in` and `out` assembly language instructions.

Minimal support
> The kernel does not recognize the hardware device, but does recognize its I/O interface. User programs are able to treat the interface as a sequential device capable of reading and/or writing sequences of characters.

Extended support
> The kernel recognizes the hardware device and handles the I/O interface itself. In fact, there might not even be a device file for the device.

The most common example of the first approach, which does not rely on any kernel device driver, is how the X Window System traditionally handles the graphic display. This is quite efficient, although it constrains the X server from using the hardware interrupts issued by the I/O device. This approach also requires some additional effort to allow the X server to access the required I/O ports. As mentioned in the section "Task State Segment" in Chapter 3, the `iopl()` and `ioperm()` system calls grant a process the privilege to access I/O ports. They can be invoked only by programs having root privileges. But such programs can be made available to users by setting the *setuid* flag of the executable file (see the section "Process Credentials and Capabilities" in Chapter 20).

Recent Linux versions support several widely used graphic cards. The */dev/fb* device file provides an abstraction for the frame buffer of the graphic card and allows application software to access it without needing to know anything about the I/O ports of the graphics interface. Furthermore, the kernel supports the Direct Rendering Infrastructure (DRI) that allows application software to exploit the hardware of accelerated 3D graphics cards. In any case, the traditional do-it-yourself X Window System server is still widely adopted.

The minimal support approach is used to handle external hardware devices connected to a general-purpose I/O interface. The kernel takes care of the I/O interface by offering a device file (and thus a device driver); the application program handles the external hardware device by reading and writing the device file.

Minimal support is preferable to extended support because it keeps the kernel size small. However, among the general-purpose I/O interfaces commonly found on a

PC, only the serial port and the parallel port can be handled with this approach. Thus, a serial mouse is directly controlled by an application program, such as the X server, and a serial modem always requires a communication program, such as Minicom, Seyon, or a Point-to-Point Protocol (PPP) daemon.

Minimal support has a limited range of applications, because it cannot be used when the external device must interact heavily with internal kernel data structures. For example, consider a removable hard disk that is connected to a general-purpose I/O interface. An application program cannot interact with all kernel data structures and functions needed to recognize the disk and to mount its filesystem, so extended support is mandatory in this case.

In general, every hardware device directly connected to the I/O bus, such as the internal hard disk, is handled according to the extended support approach: the kernel must provide a device driver for each such device. External devices attached to the Universal Serial Bus (USB), the PCMCIA port found in many laptops, or the SCSI interface—in short, every general-purpose I/O interface except the serial and the parallel ports—also require extended support.

It is worth noting that the standard file-related system calls such as `open( )`, `read( )`, and `write( )` do not always give the application full control of the underlying hardware device. In fact, the lowest-common-denominator approach of the VFS does not include room for special commands that some devices need or let an application check whether the device is in a specific internal state.

The `ioctl( )` system call was introduced to satisfy such needs. Besides the file descriptor of the device file and a second 32-bit parameter specifying the request, the system call can accept an arbitrary number of additional parameters. For example, specific `ioctl( )` requests exist to get the CD-ROM sound volume or to eject the CD-ROM media. Application programs may provide the user interface of a CD player using these kinds of `ioctl( )` requests.

Character Device Drivers

Handling a character device is relatively easy, because usually sophisticated buffering strategies are not needed and disk caches are not involved. Of course, character devices differ in their requirements: some of them must implement a sophisticated communication protocol to drive the hardware device, while others just have to read a few values from a couple of I/O ports of the hardware devices. For instance, the device driver of a multiport serial card device (a hardware device offering many serial ports) is much more complicated than the device driver of a bus mouse.

Block device drivers, on the other hand, are inherently more complex than character device drivers. In fact, applications are entitled to ask repeatedly to read or write the same block of data. Furthermore, accesses to these devices are usually very slow. These peculiarities have a profound impact on the structure of the disk drivers. As

we'll see in the next chapters, however, the kernel provides sophisticated components—such as the page cache and the block I/O subsystem—to handle them. In the rest of this chapter we focus our attention on the character device drivers.

A character device driver is described by a cdev structure, whose fields are listed in Table 13-8.

Table 13-8. The fields of the cdev structure

Type	Field	Description
struct kobject	kobj	Embedded kobject
struct module *	owner	Pointer to the module implementing the driver, if any
struct file_operations *	ops	Pointer to the file operations table of the device driver
struct list_head	list	Head of the list of inodes relative to device files for this character device
dev_t	dev	Initial major and minor numbers assigned to the device driver
unsigned int	count	Size of the range of device numbers assigned to the device driver

The list field is the head of a doubly linked circular list collecting inodes of character device files that refer to the same character device driver. There could be many device files having the same device number, and all of them refer to the same character device. Moreover, a device driver can be associated with a range of device numbers, not just a single one; all device files whose numbers fall in the range are handled by the same character device driver. The size of the range is stored in the count field.

The cdev_alloc() function allocates dynamically a cdev descriptor and initializes the embedded kobject so that the descriptor is automatically freed when the reference counter becomes zero.

The cdev_add() function registers a cdev descriptor in the device driver model. The function initializes the dev and count fields of the cdev descriptor, then invokes the kobj_map() function. This function, in turn, sets up the device driver model's data structures that glue the interval of device numbers to the device driver descriptor.

The device driver model defines a *kobject mapping domain* for the character devices, which is represented by a descriptor of type kobj_map and is referenced by the cdev_map global variable. The kobj_map descriptor includes a hash table of 255 entries indexed by the major number of the intervals. The hash table stores objects of type probe, one for each registered range of major and minor numbers, whose fields are listed in Table 13-9.

Table 13-9. The fields of the probe object

Type	Field	Description
struct probe *	next	Next element in hash collision list
dev_t	dev	Initial device number (major and minor) of the interval

Table 13-9. The fields of the probe object (continued)

Type	Field	Description
`unsigned long`	`range`	Size of the interval
`struct module *`	`owner`	Pointer to the module that implements the device driver, if any
`struct kobject *(*)` `(dev_t, int *, void *)`	`get`	Method for probing the owner of the interval
`int (*)(dev_t, void *)`	`lock`	Method for increasing the reference counter of the owner of the interval
`void *`	`data`	Private data for the owner of the interval

When the `kobj_map()` function is invoked, the specified interval of device numbers is added to the hash table. The `data` field of the corresponding `probe` object points to the `cdev` descriptor of the device driver. The value of this field is passed to the `get` and `lock` methods when they are executed. In this case, the `get` method is implemented by a short function that returns the address of the kobject embedded in the `cdev` descriptor; the `lock` method, instead, essentially increases the reference counter in the embedded kobject.

The `kobj_lookup()` function receives as input parameters a kobject mapping domain and a device number; it searches the hash table and returns the address of the kobject of the owner of the interval including the number, if it was found. When applied to the mapping domain of the character devices, the function returns the address of the kobject embedded in the `cdev` descriptor of the device driver that owns the interval of device numbers.

Assigning Device Numbers

To keep track of which character device numbers are currently assigned, the kernel uses a hash table `chrdevs`, which contains intervals of device numbers. Two intervals may share the same major number, but they cannot overlap, thus their minor numbers should be all different. The table includes 255 entries, and the hash function masks out the four higher-order bits of the major number—therefore, major numbers less than 255 are hashed in different entries. Each entry points to the first element of a collision list ordered by increasing major and minor numbers.

Each list element is a `char_device_struct` structure, whose fields are shown in Table 13-10.

Table 13-10. The fields of the char_device_struct descriptor

Type	Field	Description
`unsigned char_device_struct *`	`next`	The pointer to next element in hash collision list
`unsigned int`	`major`	The major number of the interval

Type	Field	Description
unsigned int	baseminor	The initial minor number of the interval
int	minorct	The interval size
const char *	name	The name of the device driver that handles the interval
struct file_operations *	fops	Not used
struct cdev *	cdev	Pointer to the character device driver descriptor

There are essentially two methods for assigning a range of device numbers to a character device driver. The first method, which should be used for all new device drivers, relies on the register_chrdev_region() and alloc_chrdev_region() functions, and assigns an arbitrary range of device numbers. For instance, to get an interval of numbers starting from the dev_t value dev and of size size:

```
register_chrdev_region(dev, size, "foo");
```

These functions do not execute cdev_add(), so the device driver must execute cdev_add() after the requested interval has been successfully assigned.

The second method makes use of the register_chrdev() function and assigns a fixed interval of device numbers including a single major number and minor numbers from 0 to 255. In this case, the device driver must not invoke cdev_add().

The register_chrdev_region() and alloc_chrdev_region() functions

The register_chrdev_region() function receives three parameters: the initial device number (major and minor numbers), the size of the requested range of device numbers (as the number of minor numbers), and the name of the device driver that is requesting the device numbers. The function checks whether the requested range spans several major numbers and, if so, determines the major numbers and the corresponding intervals that cover the whole range; then, the function invokes __register_chrdev_region() (described below) on each of these intervals.

The alloc_chrdev_region() function is similar, but it is used to allocate dynamically a major number; thus, it receives as its parameters the initial minor number of the interval, the size of the interval, and the name of the device driver. This function also ends up invoking __register_chrdev_region().

The __register_chrdev_region() function executes the following steps:

1. Allocates a new char_device_struct structure, and fills it with zeros.

2. If the major number of the interval is zero, then the device driver has requested the dynamic allocation of the major number. Starting from the last hash table entry and proceeding backward, the function looks for an empty collision list

(NULL pointer), which corresponds to a yet unused major number. If no empty entry is found, the function returns an error code.*

3. Initializes the fields of the `char_device_struct` structure with the initial device number of the interval, the interval size, and the name of the device driver.

4. Executes the hash function to compute the hash table index corresponding to the major number.

5. Walks the collision list, looking for the correct position of the new `char_device_struct` structure. Meanwhile, if an interval overlapping with the requested one is found, it returns an error code.

6. Inserts the new `char_device_struct` descriptor in the collision list.

7. Returns the address of the new `char_device_struct` descriptor.

The register_chrdev() function

The `register_chrdev()` function is used by drivers that require an old-style interval of device numbers: a single major number and minor numbers ranging from 0 to 255. The function receives as its parameters the requested major number `major` (zero for dynamic allocation), the name of the device driver `name`, and a pointer `fops` to a table of file operations specific to the character device files in the interval. It executes the following operations:

1. Invokes the `__register_chrdev_region()` function to allocate the requested interval. If the function returns an error code (the interval cannot be assigned), it terminates.

2. Allocates a new `cdev` structure for the device driver.

3. Initializes the `cdev` structure:

   a. Sets the type of the embedded kobject to the `ktype_cdev_dynamic` type descriptor (see the earlier section "Kobjects").

   b. Sets the `owner` field with the contents of `fops->owner`.

   c. Sets the `ops` field with the address `fops` of the table of file operations.

   d. Copies the characters of the device driver name into the `name` field of the embedded kobject.

4. Invokes the `cdev_add()` function (explained previously).

---

* Notice that the kernel can dynamically allocate only major numbers less than 255, and that in some cases allocation can fail even if there is a unused major number less than 255. We might expect that these constraints will be removed in the future.

---

5. Sets the `cdev` field of the `char_device_struct` descriptor `__register_chrdev_region()` returned in step 1 with the address of the `cdev` descriptor of the device driver.

6. Returns the major number of the assigned interval.

Accessing a Character Device Driver

We mentioned in the earlier section "VFS Handling of Device Files" that the `dentry_open()` function triggered by the `open()` system call service routine customizes the `f_op` field in the file object of the character device file so that it points to the `def_chr_fops` table. This table is almost empty; it only defines the `chrdev_open()` function as the open method of the device file. This method is immediately invoked by `dentry_open()`.

The `chrdev_open()` function receives as its parameters the addresses `inode` and `filp` of the inode and file objects relative to the device file being opened. It executes essentially the following operations:

1. Checks the `inode->i_cdev` pointer to the device driver's `cdev` descriptor. If this field is not `NULL`, then the inode has already been accessed: increases the reference counter of the `cdev` descriptor and jumps to step 6.

2. Invokes the `kobj_lookup()` function to search the interval including the number. If such interval does not exists, it returns an error code; otherwise, it computes the address of the `cdev` descriptor associated with the interval.

3. Sets the `inode->i_cdev` field of the inode object to the address of the `cdev` descriptor.

4. Sets the `inode->i_cindex` field to the relative index of the device number inside the interval of the device driver (index zero for the first minor number in the interval, one for the second, and so on).

5. Adds the inode object into the list pointed to by the `list` field of the `cdev` descriptor.

6. Initializes the `filp->f_ops` file operations pointer with the contents of the `ops` field of the `cdev` descriptor.

7. If the `filp->f_ops->open` method is defined, the function executes it. If the device driver handles more than one device number, typically this function sets the file operations of the file object once again, so as to install the file operations suitable for the accessed device file.

8. Terminates by returning zero (success).

Buffering Strategies for Character Devices

Traditionally, Unix-like operating systems divide hardware devices into block and character devices. However, this classification does not tell the whole story. Some devices are capable of transferring sizeable amounts of data in a single I/O operation, while others transfer only a few characters.

For instance, a PS/2 mouse driver gets a few bytes in each read operation corresponding to the status of the mouse button and to the position of the mouse pointer on the screen. This kind of device is the easiest to handle. Input data is first read one character at a time from the device input register and stored in a proper kernel data structure; the data is then copied at leisure into the process address space. Similarly, output data is first copied from the process address space to a proper kernel data structure and then written one at a time into the I/O device output register. Clearly, I/O drivers for such devices do not use the DMA, because the CPU time spent to set up a DMA I/O operation is comparable to the time spent to move the data to or from the I/O ports.

On the other hand, the kernel must also be ready to deal with devices that yield a large number of bytes in each I/O operation, either sequential devices such as sound cards or network cards, or random access devices such as disks of all kinds (floppy, CD-ROM, SCSI disk, etc.).

Suppose, for instance, that you have set up the sound card of your computer so that you are able to record sounds coming from a microphone. The sound card samples the electrical signal coming from the microphone at a fixed rate, say 44.14 kHz, and produces a stream of 16-bit numbers divided into blocks of input data. The sound card driver must be able to cope with this avalanche of data in all possible situations, even when the CPU is temporarily busy running some other process.

This can be done by combining two different techniques:

- Use of DMA to transfer blocks of data.
- Use of a circular buffer of two or more elements, each element having the size of a block of data. When an interrupt occurs signaling that a new block of data has been read, the interrupt handler advances a pointer to the elements of the circular buffer so that further data will be stored in an empty element. Conversely, whenever the driver succeeds in copying a block of data into user address space, it releases an element of the circular buffer so that it is available for saving new data from the hardware device.

The role of the circular buffer is to smooth out the peaks of CPU load; even if the User Mode application receiving the data is slowed down because of other higher-priority tasks, the DMA is able to continue filling elements of the circular buffer because the interrupt handler executes on behalf of the currently running process.

A similar situation occurs when receiving packets from a network card, except that in this case, the flow of incoming data is asynchronous. Packets are received independently from each other and the time interval that occurs between two consecutive packet arrivals is unpredictable.

All considered, buffer handling for sequential devices is easy because the same buffer is *never reused*: an audio application cannot ask the microphone to retransmit the same block of data.

We'll see in Chapter 15 that buffering for random access devices (all kinds of disks) is much more complicated.

CHAPTER 14

Block Device Drivers

This chapter deals with I/O drivers for block devices, i.e., for disks of every kind. The key aspect of a block device is the disparity between the time taken by the CPU and buses to read or write data and the speed of the disk hardware. Block devices have very high average access times. Each operation requires several milliseconds to complete, mainly because the disk controller must move the heads on the disk surface to reach the exact position where the data is recorded. However, when the heads are correctly placed, data transfer can be sustained at rates of tens of megabytes per second.

The organization of Linux block device handlers is quite involved. We won't be able to discuss in detail all the functions that are included in the block I/O subsystem of the kernel; however, we'll outline the general software architecture. As in the previous chapter, our objective is to explain how Linux supports the implementation of block device drivers, rather than showing how to implement one of them.

We start in the first section "Block Devices Handling" to explain the general architecture of the Linux block I/O subsystem. In the sections "The Generic Block Layer," "The I/O Scheduler," and "Block Device Drivers," we will describe the main components of the block I/O subsystem. Finally, in the last section, "Opening a Block Device File," we will outline the steps performed by the kernel when opening a block device file.

Block Devices Handling

Each operation on a block device driver involves a large number of kernel components; the most important ones are shown in Figure 14-1.

Let us suppose, for instance, that a process issued a `read()` system call on some disk file—we'll see that write requests are handled essentially in the same way. Here is what the kernel typically does to service the process request:

1. The service routine of the `read()` system call activates a suitable VFS function, passing to it a file descriptor and an offset inside the file. The Virtual Filesystem
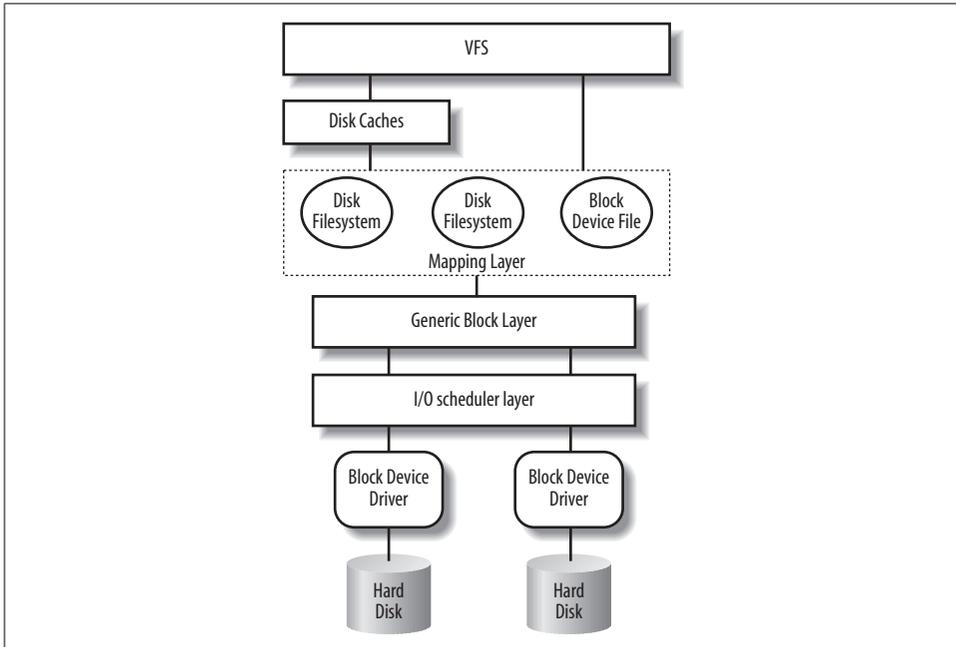
Figure 14-1. Kernel components affected by a block device operation

is the upper layer of the block device handling architecture, and it provides a common file model adopted by all filesystems supported by Linux. We have described at length the VFS layer in Chapter 12.

2. The VFS function determines if the requested data is already available and, if necessary, how to perform the read operation. Sometimes there is no need to access the data on disk, because the kernel keeps in RAM the data most recently read from—or written to—a block device. The disk cache mechanism is explained in Chapter 15, while details on how the VFS handles the disk operations and how it interfaces with the disk cache and the filesystems are given in Chapter 16.

3. Let's assume that the kernel must read the data from the block device, thus it must determine the physical location of that data. To do this, the kernel relies on the *mapping layer*, which typically executes two steps:

   a. It determines the block size of the filesystem including the file and computes the extent of the requested data in terms of *file block numbers*. Essentially, the file is seen as split in many blocks, and the kernel determines the numbers (indices relative to the beginning of file) of the blocks containing the requested data.

   b. Next, the mapping layer invokes a filesystem-specific function that accesses the file's disk inode and determines the position of the requested data on disk in terms of *logical block numbers*. Essentially, the disk is seen as split in

blocks, and the kernel determines the numbers (indices relative to the beginning of the disk or partition) corresponding to the blocks storing the requested data. Because a file may be stored in nonadjacent blocks on disk, a data structure stored in the disk inode maps each file block number to a logical block number.[*]

We will see the mapping layer in action in Chapter 16, while we will present some typical disk-based filesystems in Chapter 18.

4. The kernel can now issue the read operation on the block device. It makes use of the *generic block layer*, which starts the I/O operations that transfer the requested data. In general, each I/O operation involves a group of blocks that are adjacent on disk. Because the requested data is not necessarily adjacent on disk, the generic block layer might start several I/O operations. Each I/O operation is represented by a "block I/O" (in short, "bio") structure, which collects all information needed by the lower components to satisfy the request.

The generic block layer hides the peculiarities of each hardware block device, thus offering an abstract view of the block devices. Because almost all block devices are disks, the generic block layer also provides some general data structures that describe "disks" and "disk partitions." We will discuss the generic block layer and the bio structure in the section "The Generic Block Layer" later in this chapter.

5. Below the generic block layer, the "I/O scheduler" sorts the pending I/O data transfer requests according to predefined kernel policies. The purpose of the scheduler is to group requests of data that lie near each other on the physical medium. We will describe this component in the section "The I/O Scheduler" later in this chapter.

6. Finally, the *block device drivers* take care of the actual data transfer by sending suitable commands to the hardware interfaces of the disk controllers. We will explain the overall organization of a generic block device driver in the section "Block Device Drivers" later in this chapter.

As you can see, there are many kernel components that are concerned with data stored in block devices; each of them manages the disk data using chunks of different length:

- The controllers of the hardware block devices transfer data in chunks of fixed length called "sectors." Therefore, the I/O scheduler and the block device drivers must manage sectors of data.

---

[*] However, if the read access was done on a raw block device file, the mapping layer does not invoke a filesystem-specific method; rather, it translates the offset in the block device file to a position inside the disk—or disk partition—corresponding to the device file.

- The Virtual Filesystem, the mapping layer, and the filesystems group the disk data in logical units called "blocks." A block corresponds to the minimal disk storage unit inside a filesystem.

- As we will see shortly, block device drivers should be able to cope with "segments" of data: each segment is a memory page—or a portion of a memory page—including chunks of data that are physically adjacent on disk.

- The disk caches work on "pages" of disk data, each of which fits in a page frame.

- The generic block layer glues together all the upper and lower components, thus it knows about sectors, blocks, segments, and pages of data.

Even if there are many different chunks of data, they usually share the same physical RAM cells. For instance, Figure 14-2 shows the layout of a 4,096-byte page. The upper kernel components see the page as composed of four block buffers of 1,024 bytes each. The last three blocks of the page are being transferred by the block device driver, thus they are inserted in a segment covering the last 3,072 bytes of the page. The hard disk controller considers the segment as composed of six 512-byte sectors.
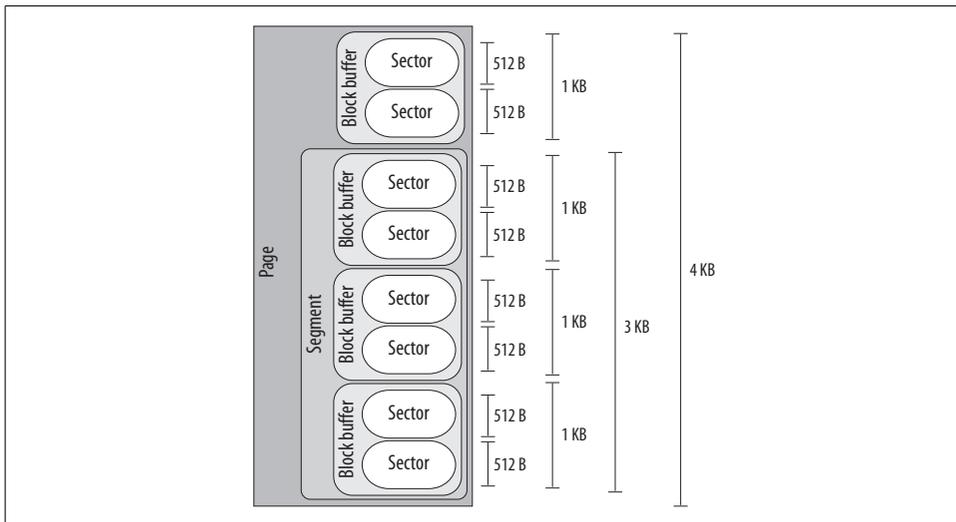
Figure 14-2. Typical layout of a page including disk data

In this chapter we describe the lower kernel components that handle the block devices—generic block layer, I/O scheduler, and block device drivers—thus we focus our attention on sectors, blocks, and segments.

Sectors

To achieve acceptable performance, hard disks and similar devices transfer several adjacent bytes at once. Each data transfer operation for a block device acts on a group of adjacent bytes called a *sector*. In the following discussion, we say that

groups of bytes are *adjacent* when they are recorded on the disk surface in such a manner that a single seek operation can access them. Although the physical geometry of a disk is usually very complicated, the hard disk controller accepts commands that refer to the disk as a large array of sectors.

In most disk devices, the size of a sector is 512 bytes, although there are devices that use larger sectors (1,024 and 2,048 bytes). Notice that the sector should be considered as the basic unit of data transfer; it is never possible to transfer less than one sector, although most disk devices are capable of transferring several adjacent sectors at once.

In Linux, the size of a sector is conventionally set to 512 bytes; if a block device uses larger sectors, the corresponding low-level block device driver will do the necessary conversions. Thus, a group of data stored in a block device is identified on disk by its position—the index of the first 512-byte sector—and its length as number of 512-byte sectors. Sector indices are stored in 32- or 64-bit variables of type `sector_t`.

Blocks

While the sector is the basic unit of data transfer for the hardware devices, the *block* is the basic unit of data transfer for the VFS and, consequently, for the filesystems. For example, when the kernel accesses the contents of a file, it must first read from disk a block containing the disk inode of the file (see the section "Inode Objects" in Chapter 12). This block on disk corresponds to one or more adjacent sectors, which are looked at by the VFS as a single data unit.

In Linux, the block size must be a power of 2 and cannot be larger than a page frame. Moreover, it must be a multiple of the sector size, because each block must include an integral number of sectors. Therefore, on 80×86 architecture, the permitted block sizes are 512, 1,024, 2,048, and 4,096 bytes.

The block size is not specific to a block device. When creating a disk-based filesystem, the administrator may select the proper block size. Thus, several partitions on the same disk might make use of different block sizes. Furthermore, each read or write operation issued on a block device file is a "raw" access that bypasses the disk-based filesystem; the kernel executes it by using blocks of largest size (4,096 bytes).

Each block requires its own *block buffer*, which is a RAM memory area used by the kernel to store the block's content. When the kernel reads a block from disk, it fills the corresponding block buffer with the values obtained from the hardware device; similarly, when the kernel writes a block on disk, it updates the corresponding group of adjacent bytes on the hardware device with the actual values of the associated block buffer. The size of a block buffer always matches the size of the corresponding block.

Each buffer has a "buffer head" descriptor of type `buffer_head`. This descriptor contains all the information needed by the kernel to know how to handle the buffer; thus, before operating on each buffer, the kernel checks its buffer head. We will give a

detailed explanation of all fields of the buffer head in Chapter 15; in the present chapter, however, we will only consider a few fields: b_page, b_data, b_blocknr, and b_bdev.

The b_page field stores the page descriptor address of the page frame that includes the block buffer. If the page frame is in high memory, the b_data field stores the offset of the block buffer inside the page; otherwise, it stores the starting linear address of the block buffer itself. The b_blocknr field stores the logical block number (i.e., the index of the block inside the disk partition). Finally, the b_bdev field identifies the block device that is using the buffer head (see the section "Block Devices" later in this chapter).

Segments

We know that each disk I/O operation consists of transferring the contents of some adjacent sectors from—or to—some RAM locations. In almost all cases, the data transfer is directly performed by the disk controller with a DMA operation (see the section "Direct Memory Access (DMA)" in Chapter 13). The block device driver simply triggers the data transfer by sending suitable commands to the disk controller; once the data transfer is finished, the controller raises an interrupt to notify the block device driver.

The data transferred by a single DMA operation must belong to sectors that are adjacent on disk. This is a physical constraint: a disk controller that allows DMA transfers to non-adjacent sectors would have a poor transfer rate, because moving a read/write head on the disk surface is quite a slow operation.

Older disk controllers support "simple" DMA operations only: in each such operation, data is transferred from or to memory cells that are physically contiguous in RAM. Recent disk controllers, however, may also support the so-called *scatter-gather DMA transfers*: in each such operation, the data can be transferred from or to several noncontiguous memory areas.

For each scatter-gather DMA transfer, the block device driver must send to the disk controller:

- The initial disk sector number and the total number of sectors to be transferred
- A list of descriptors of memory areas, each of which consists of an address and a length.

The disk controller takes care of the whole data transfer; for instance, in a read operation the controller fetches the data from the adjacent disk sectors and scatters it into the various memory areas.

To make use of scatter-gather DMA operations, block device drivers must handle the data in units called *segments*. A segment is simply a memory page—or a portion of a memory page—that includes the data of some adjacent disk sectors. Thus, a scatter-gather DMA operation may involve several segments at once.

Notice that a block device driver does not need to know about blocks, block sizes, and block buffers. Thus, even if a segment is seen by the higher levels as a page composed of several block buffers, the block device driver does not care about it.

As we'll see, the generic block layer can merge different segments if the corresponding page frames happen to be contiguous in RAM and the corresponding chunks of disk data are adjacent on disk. The larger memory area resulting from this merge operation is called *physical segment*.

Yet another merge operation is allowed on architectures that handle the mapping between bus addresses and physical addresses through a dedicated bus circuitry (the IO-MMU; see the section "Direct Memory Access (DMA)" in Chapter 13). The memory area resulting from this kind of merge operation is called *hardware segment*. Because we will focus on the 80×86 architecture, which has no such dynamic mapping between bus addresses and physical addresses, we will assume in the rest of this chapter that hardware segments always coincide with physical segments.

The Generic Block Layer

The generic block layer is a kernel component that handles the requests for all block devices in the system. Thanks to its functions, the kernel may easily:

- Put data buffers in high memory—the page frame(s) will be mapped in the kernel linear address space only when the CPU must access the data, and will be unmapped right after.

- Implement—with some additional effort—a "zero-copy" schema, where disk data is directly put in the User Mode address space without being copied to kernel memory first; essentially, the buffer used by the kernel for the I/O transfer lies in a page frame mapped in the User Mode linear address space of a process.

- Manage logical volumes—such as those used by LVM (the Logical Volume Manager) and RAID (Redundant Array of Inexpensive Disks): several disk partitions, even on different block devices, can be seen as a single partition.

- Exploit the advanced features of the most recent disk controllers, such as large onboard disk caches, enhanced DMA capabilities, onboard scheduling of the I/O transfer requests, and so on.

The Bio Structure

The core data structure of the generic block layer is a descriptor of an ongoing I/O block device operation called *bio*. Each bio essentially includes an identifier for a disk storage area—the initial sector number and the number of sectors included in the storage area—and one or more segments describing the memory areas involved in the I/O operation. A bio is implemented by the `bio` data structure, whose fields are listed in Table 14-1.

Table 14-1. The fields of the bio structure

Type	Field	Description
sector_t	bi_sector	First sector on disk of block I/O operation
struct bio *	bi_next	Link to the next bio in the request queue
struct block_device *	bi_bdev	Pointer to block device descriptor
unsigned long	bi_flags	Bio status flags
unsigned long	bi_rw	I/O operation flags
unsigned short	bi_vcnt	Number of segments in the bio's bio_vec array
unsigned short	bi_idx	Current index in the bio's bio_vec array of segments
unsigned short	bi_phys_segments	Number of physical segments of the bio after merging
unsigned short	bi_hw_segments	Number of hardware segments after merging
unsigned int	bi_size	Bytes (yet) to be transferred
unsigned int	bi_hw_front_size	Used by the hardware segment merge algorithm
unsigned int	bi_hw_back_size	Used by the hardware segment merge algorithm
unsigned int	bi_max_vecs	Maximum allowed number of segments in the bio's bio_vec array
struct bio_vec *	bi_io_vec	Pointer to the bio's bio_vec array of segments
bio_end_io_t *	bi_end_io	Method invoked at the end of bio's I/O operation
atomic_t	bi_cnt	Reference counter for the bio
void *	bi_private	Pointer used by the generic block layer and the I/O completion method of the block device driver
bio_destructor_t *	bi_destructor	Destructor method (usually bio_destructor()) invoked when the bio is being freed

Each segment in a bio is represented by a bio_vec data structure, whose fields are listed in Table 14-2. The bi_io_vec field of the bio points to the first element of an array of bio_vec data structures, while the bi_vcnt field stores the current number of elements in the array.

Table 14-2. The fields of the bio_vec structure

Type	Field	Description
struct page *	bv_page	Pointer to the page descriptor of the segment's page frame
unsigned int	bv_len	Length of the segment in bytes
unsigned int	bv_offset	Offset of the segment's data in the page frame

The contents of a bio descriptor keep changing during the block I/O operation. For instance, if the block device driver cannot perform the whole data transfer with one scatter-gather DMA operation, the bi_idx field is updated to keep track of the first segment in the bio that is yet to be transferred. To iterate over the segments of a bio—starting from the current segment at index bi_idx—a device driver can execute the bio_for_each_segment macro.

When the generic block layer starts a new I/O operation, it allocates a new bio structure by invoking the `bio_alloc()` function. Usually, bios are allocated through the slab allocator, but the kernel also keeps a small memory pool of bios to be used when memory is scarce (see the section "Memory Pools" in Chapter 8). The kernel also keeps a memory pool for the `bio_vec` structures—after all, it would not make sense to allocate a bio without being able to allocate the segment descriptors to be included in the bio. Correspondingly, the `bio_put()` function decrements the reference counter (`bi_cnt`) of a bio and, if the counter becomes zero, it releases the bio structure and the related `bio_vec` structures.

Representing Disks and Disk Partitions

A *disk* is a logical block device that is handled by the generic block layer. Usually a disk corresponds to a hardware block device such as a hard disk, a floppy disk, or a CD-ROM disk. However, a disk can be a virtual device built upon several physical disk partitions, or a storage area living in some dedicated pages of RAM. In any case, the upper kernel components operate on all disks in the same way thanks to the services offered by the generic block layer.

A disk is represented by the `gendisk` object, whose fields are shown in Table 14-3.

Table 14-3. The fields of the gendisk object

Type	Field	Description
int	major	Major number of the disk
int	first_minor	First minor number associated with the disk
int	minors	Range of minor numbers associated with the disk
char [32]	disk_name	Conventional name of the disk (usually, the canonical name of the corresponding device file)
struct hd_struct **	part	Array of partition descriptors for the disk
struct block_device_operations *	fops	Pointer to a table of block device methods
struct request_queue *	queue	Pointer to the request queue of the disk (see "Request Queue Descriptors" later in this chapter)
void *	private_data	Private data of the block device driver
sector_t	capacity	Size of the storage area of the disk (in number of sectors)
int	flags	Flags describing the kind of disk (see below)
char [64]	devfs_name	Device filename in the (nowadays deprecated) *devfs* special filesystem
int	number	No longer used
struct device *	driverfs_dev	Pointer to the `device` object of the disk's hardware device (see the section "Components of the Device Driver Model" in Chapter 13)
struct kobject	kobj	Embedded kobject (see the section "Kobjects" in Chapter 13)

Table 14-3. The fields of the gendisk object (continued)

Type	Field	Description
struct timer_rand_state *	random	Pointer to a data structure that records the timing of the disk's interrupts; used by the kernel built-in random number generator
int	policy	Set to 1 if the disk is read-only (write operations forbidden), 0 otherwise
atomic_t	sync_io	Counter of sectors written to disk, used only for RAID
unsigned long	stamp	Timestamp used to determine disk queue usage statistics
unsigned long	stamp_idle	Same as above
int	in_flight	Number of ongoing I/O operations
struct disk_stats *	dkstats	Statistics about per-CPU disk usage

The `flags` field stores information about the disk. The most important flag is `GENHD_FL_UP`: if it is set, the disk is initialized and working. Another relevant flag is `GENHD_FL_REMOVABLE`, which is set if the disk is a removable support, such as a floppy disk or a CD-ROM.

The `fops` field of the `gendisk` object points to a `block_device_operations` table, which stores a few custom methods for crucial operations of the block device (see Table 14-4).

Table 14-4. The methods of the block devices

Method	Triggers
open	Opening the block device file
release	Closing the last reference to a block device file
ioctl	Issuing an `ioctl()` system call on the block device file (uses the big kernel lock)
compat_ioctl	Issuing an `ioctl()` system call on the block device file (does not use the big kernel lock)
media_changed	Checking whether the removable media has been changed (e.g., floppy disk)
revalidate_disk	Checking whether the block device holds valid data

Hard disks are commonly split into logical *partitions*. Each block device file may represent either a whole disk or a partition inside the disk. For instance, a master EIDE disk might be represented by a device file */dev/hda* having major number 3 and minor number 0; the first two partitions inside the disk might be represented by device files */dev/hda1* and */dev/hda2* having major number 3 and minor numbers 1 and 2, respectively. In general, the partitions inside a disk are characterized by consecutive minor numbers.

If a disk is split in partitions, their layout is kept in an array of `hd_struct` structures whose address is stored in the `part` field of the `gendisk` object. The array is indexed

by the relative index of the partition inside the disk. The fields of the hd_struct descriptor are listed in Table 14-5.

Table 14-5. The fields of the hd_struct descriptor

Type	Field	Description
sector_t	start_sect	Starting sector of the partition inside the disk
sector_t	nr_sects	Length of the partition (number of sectors)
struct kobject	kobj	Embedded kobject (see the section "Kobjects" in Chapter 13)
unsigned int	reads	Number of read operations issued on the partition
unsigned int	read_sectors	Number of sectors read from the partition
unsigned int	writes	Number of write operations issued on the partition
unsigned int	write_sectors	Number of sectors written into the partition
int	policy	Set to 1 if the partition is read-only, 0 otherwise
int	partno	The relative index of the partition inside the disk

When the kernel discovers a new disk in the system (in the boot phase, or when a removable media is inserted in a drive, or when an external disk is attached at runtime), it invokes the alloc_disk() function, which allocates and initializes a new gendisk object and, if the new disk is split in several partitions, a suitable array of hd_struct descriptors. Then, it invokes the add_disk() function to insert the new gendisk descriptor into the data structures of the generic block layer (see the section "Device Driver Registration and Initialization" later in this chapter).

Submitting a Request

Let us describe the common sequence of steps executed by the kernel when submitting an I/O operation request to the generic block layer. We'll assume that the requested chunks of data are adjacent on disk and that the kernel has already determined their physical location.

The first step consists in executing the bio_alloc() function to allocate a new bio descriptor. Then, the kernel initializes the bio descriptor by setting a few fields:

- The bi_sector field is set to the initial sector number of the data (if the block device is split in several partitions, the sector number is relative to the start of the partition).
- The bi_size field is set to the number of sectors covering the data.
- The bi_bdev field is set to the address of the block device descriptor (see the section "Block Devices" later in this chapter).
- The bi_io_vec field is set to the initial address of an array of bio_vec data structures, each of which describes a segment (memory buffer) involved in the I/O operation; moreover, the bi_vcnt field is set to the total number of segments in the bio.

- The `bi_rw` field is set with the flags of the requested operation. The most important flag specifies the data transfer direction: READ (0) or WRITE (1).
- The `bi_end_io` field is set to the address of a completion procedure that is executed whenever the I/O operation on the bio is completed.

Once the `bio` descriptor has been properly initialized, the kernel invokes the `generic_make_request()` function, which is the main entry point of the generic block layer. The function essentially executes the following steps:

1. Checks that `bio->bi_sector` does not exceed the number of sectors of the block device. If it does, the function sets the `BIO_EOF` flag of `bio->bi_flags`, prints a kernel error message, invokes the `bio_endio()` function, and terminates. `bio_endio()` updates the `bi_size` and `bi_sector` fields of the bio descriptor, and it invokes the `bi_end_io` bio's method. The implementation of the latter function essentially depends on the kernel component that has triggered the I/O data transfer; we will see some examples of `bi_end_io` methods in the following chapters.

2. Gets the request queue q associated with the block device (see the section "Request Queue Descriptors" later in this chapter); its address can be found in the `bd_disk` field of the block device descriptor, which in turn is pointed to by the `bio->bi_bdev` field.

3. Invokes `block_wait_queue_running()` to check whether the I/O scheduler currently in use is being dynamically replaced; in this case, the function puts the process to sleep until the new I/O scheduler is started (see the next section "The I/O Scheduler").

4. Invokes `blk_partition_remap()` to check whether the block device refers to a disk partition (`bio->bi_bdev` not equal to `bio->bi_dev->bd_contains`; see the section "Block Devices" later in this chapter). In this case, the function gets the `hd_struct` descriptor of the partition from the `bio->bi_bdev` field to perform the following substeps:

   a. Updates the `read_sectors` and `reads` fields, or the `write_sectors` and `writes` fields, of the `hd_struct` descriptor, according to the direction of data transfer.

   b. Adjusts the `bio->bi_sector` field so as to transform the sector number relative to the start of the partition to a sector number relative to the whole disk.

   c. Sets the `bio->bi_bdev` field to the block device descriptor of the whole disk (`bio->bd_contains`).

   From now on, the generic block layer, the I/O scheduler, and the device driver forget about disk partitioning, and work directly with the whole disk.

5. Invokes the `q->make_request_fn` method to insert the bio request in the request queue q.

6. Returns.

We will discuss a typical implementation of the make_request_fn method in the section "Issuing a Request to the I/O Scheduler" later in this chapter.

The I/O Scheduler

Although block device drivers are able to transfer a single sector at a time, the block I/O layer does not perform an individual I/O operation for each sector to be accessed on disk; this would lead to poor disk performance, because locating the physical position of a sector on the disk surface is quite time-consuming. Instead, the kernel tries, whenever possible, to cluster several sectors and handle them as a whole, thus reducing the average number of head movements.

When a kernel component wishes to read or write some disk data, it actually creates a *block device request*. That request essentially describes the requested sectors and the kind of operation to be performed on them (read or write). However, the kernel does not satisfy a request as soon as it is created—the I/O operation is just scheduled and will be performed at a later time. This artificial delay is paradoxically the crucial mechanism for boosting the performance of block devices. When a new block data transfer is requested, the kernel checks whether it can be satisfied by slightly enlarging a previous request that is still waiting (i.e., whether the new request can be satisfied without further seek operations). Because disks tend to be accessed sequentially, this simple mechanism is very effective.

Deferring requests complicates block device handling. For instance, suppose a process opens a regular file and, consequently, a filesystem driver wants to read the corresponding inode from disk. The block device driver puts the request on a queue, and the process is suspended until the block storing the inode is transferred. However, the block device driver itself cannot be blocked, because any other process trying to access the same disk would be blocked as well.

To keep the block device driver from being suspended, each I/O operation is processed asynchronously. In particular, block device drivers are interrupt-driven (see the section "Monitoring I/O Operations" in the previous chapter): the generic block layer invokes the *I/O scheduler* to create a new block device request or to enlarge an already existing one and then terminates. The block device driver, which is activated at a later time, invokes the *strategy routine* to select a pending request and satisfy it by issuing suitable commands to the disk controller. When the I/O operation terminates, the disk controller raises an interrupt and the corresponding handler invokes the strategy routine again, if necessary, to process another pending request.

Each block device driver maintains its own *request queue*, which contains the list of pending requests for the device. If the disk controller is handling several disks, there is usually one request queue for each physical block device. I/O scheduling is performed separately on each request queue, thus increasing disk performance.

Request Queue Descriptors

Each request queue is represented by means of a large request_queue data structure whose fields are listed in Table 14-6.

Table 14-6. The fields of the request queue descriptor

Type	Field	Description
struct list_head	queue_head	List of pending requests
struct request *	last_merge	Pointer to descriptor of the request in the queue to be considered first for possible merging
elevator_t *	elevator	Pointer to the elevator object (see the later section "I/O Scheduling Algorithms")
struct request_list	rq	Data structure used for allocation of request descriptors
request_fn_proc *	request_fn	Method that implements the entry point of the strategy routine of the driver
merge_request_fn *	back_merge_fn	Method to check whether it is possible to merge a bio to the last request in the queue
merge_request_fn *	front_merge_fn	Method to check whether it is possible to merge a bio to the first request in the queue
merge_requests_fn *	merge_requests_fn	Method to attempt merging two adjacent requests in the queue
make_request_fn *	make_request_fn	Method invoked when a new request has to be inserted in the queue
prep_rq_fn *	prep_rq_fn	Method to build the commands to be sent to the hardware device to process this request
unplug_fn *	unplug_fn	Method to unplug the block device (see the section "Activating the Block Device Driver" later in the chapter)
merge_bvec_fn *	merge_bvec_fn	Method that returns the number of bytes that can be inserted into an existing bio when adding a new segment (usually undefined)
activity_fn *	activity_fn	Method invoked when a request is added to a queue (usually undefined)
issue_flush_fn *	issue_flush_fn	Method invoked when a request queue is flushed (the queue is emptied by processing all requests in a row)
struct timer_list	unplug_timer	Dynamic timer used to perform device plugging (see the later section "Activating the Block Device Driver")
int	unplug_thresh	If the number of pending requests in the queue exceeds this value, the device is immediately unplugged (default is 4)
unsigned long	unplug_delay	Time delay before device unplugging (default is 3 milliseconds)

Table 14-6. The fields of the request queue descriptor (continued)

Type	Field	Description
struct work_struct	unplug_work	Work queue used to unplug the device (see the later section "Activating the Block Device Driver")
struct backing_dev_info	backing_dev_info	See the text following this table
void *	queuedata	Pointer to private data of the block device driver
void *	activity_data	Private data used by the activity_fn method
unsigned long	bounce_pfn	Page frame number above which buffer bouncing must be used (see the section "Submitting a Request" later in this chapter)
int	bounce_gfp	Memory allocation flags for bounce buffers
unsigned long	queue_flags	Set of flags describing the queue status
spinlock_t *	queue_lock	Pointer to request queue lock
struct kobject	kobj	Embedded kobject for the request queue
unsigned long	nr_requests	Maximum number of requests in the queue
unsigned int	nr_congestion_on	Queue is considered congested if the number of pending requests rises above this threshold
unsigned int	nr_congestion_off	Queue is considered not congested if the number of pending requests falls below this threshold
unsigned int	nr_batching	Maximum number (usually 32) of pending requests that can be submitted even when the queue is full by a special "batcher" process
unsigned short	max_sectors	Maximum number of sectors handled by a single request (tunable)
unsigned short	max_hw_sectors	Maximum number of sectors handled by a single request (hardware constraint)
unsigned short	max_phys_segments	Maximum number of physical segments handled by a single request
unsigned short	max_hw_segments	Maximum number of hardware segments handled by a single request (the maximum number of distinct memory areas in a scatter-gather DMA operation)
unsigned short	hardsect_size	Size in bytes of a sector
unsigned int	max_segment_size	Maximum size of a physical segment (in bytes)
unsigned long	seg_boundary_mask	Memory boundary mask for segment merging
unsigned int	dma_alignment	Alignment bitmap for initial address and length of DMA buffers (default 511)
struct blk_queue_tag *	queue_tags	Bitmap of free/busy tags (used for tagged requests)
atomic_t	refcnt	Reference counter of the queue
unsigned int	in_flight	Number of pending requests in the queue
unsigned int	sg_timeout	User-defined command time-out (used only by SCSI generic devices)

Table 14-6. The fields of the request queue descriptor (continued)

Type	Field	Description
`unsigned int`	`sg_reserved_size`	Essentially unused
`struct list_head`	`drain_list`	Head of a list of requests temporarily delayed until the I/O scheduler is dynamically replaced

Essentially, a request queue is a doubly linked list whose elements are request descriptors (that is, request data structures; see the next section). The queue_head field of the request queue descriptor stores the head (the first dummy element) of the list, while the pointers in the `queuelist` field of the request descriptor link each request to the previous and next elements in the list. The ordering of the elements in the queue list is specific to each block device driver; the I/O scheduler offers, however, several predefined ways of ordering elements, which are discussed in the later section "I/O Scheduling Algorithms."

The `backing_dev_info` field is a small object of type `backing_dev_info`, which stores information about the I/O data flow traffic for the underlying hardware block device. For instance, it holds information about read-ahead and about request queue congestion state.

Request Descriptors

Each pending request for a block device is represented by a *request descriptor*, which is stored in the `request` data structure illustrated in Table 14-7.

Table 14-7. The fields of the request descriptor

Type	Field	Description
`struct list_head`	`queuelist`	Pointers for request queue list
`unsigned long`	`flags`	Flags of the request (see below)
`sector_t`	`sector`	Number of the next sector to be transferred
`unsigned long`	`nr_sectors`	Number of sectors yet to be transferred in the whole request
`unsigned int`	`current_nr_sectors`	Number of sectors in the current segment of the current bio yet to be transferred
`sector_t`	`hard_sector`	Number of the next sector to be transferred
`unsigned long`	`hard_nr_sectors`	Number of sectors yet to be transferred in the whole request (updated by the generic block layer)
`unsigned int`	`hard_cur_sectors`	Number of sectors in the current segment of the current bio yet to be transferred (updated by the generic block layer)
`struct bio *`	`bio`	First bio in the request that has not been completely transferred
`struct bio *`	`biotail`	Last bio in the request list

Table 14-7. The fields of the request descriptor (continued)

Type	Field	Description
void *	elevator_private	Pointer to private data for the I/O scheduler
int	rq_status	Request status: essentially, either RQ_ACTIVE or RQ_INACTIVE
struct gendisk *	rq_disk	The descriptor of the disk referenced by the request
int	errors	Counter for the number of I/O errors that occurred on the current transfer
unsigned long	start_time	Request's starting time (in jiffies)
unsigned short	nr_phys_segments	Number of physical segments of the request
unsigned short	nr_hw_segments	Number of hardware segments of the request
int	tag	Tag associated with the request (only for hardware devices supporting multiple outstanding data transfers)
char *	buffer	Pointer to the memory buffer of the current data transfer (NULL if the buffer is in high-memory)
int	ref_count	Reference counter for the request
request_queue_t *	q	Pointer to the descriptor of the request queue containing the request
struct request_list *	rl	Pointer to request_list data structure
struct completion *	waiting	Completion for waiting for the end of the data transfers (see the section "Completions" in Chapter 5)
void *	special	Pointer to data used when the request includes a "special" command to the hardware device
unsigned int	cmd_len	Length of the commands in the cmd field
unsigned char []	cmd	Buffer containing the pre-built commands prepared by the request queue's prep_rq_fn method
unsigned int	data_len	Usually, the length of data in the buffer pointed to by the data field
void *	data	Pointer used by the device driver to keep track of the data to be transferred
unsigned int	sense_len	Length of buffer pointed to by the sense field (0 if the sense field is NULL)
void *	sense	Pointer to buffer used for output of sense commands
unsigned int	timeout	Request's time-out
struct request_pm_state *	pm	Pointer to a data structure used for power-management commands

Each request consists of one or more bio structures. Initially, the generic block layer creates a request including just one bio. Later, the I/O scheduler may "extend" the request either by adding a new segment to the original bio, or by linking another bio structure into the request. This is possible when the new data is physically adjacent to the data already in the request. The `bio` field of the request descriptor points to the first bio structure in the request, while the `biotail` field points to the last bio. The `rq_for_each_bio` macro implements a loop that iterates over all bios included in a request.

Several fields of the request descriptor may dynamically change. For instance, as soon as the chunks of data referenced in a bio have all been transferred, the `bio` field is updated so that it points to the next bio in the request list. Meanwhile, new bios can be added to the tail of the request list, so the `biotail` field may also change.

Several other fields of the request descriptor are modified either by the I/O scheduler or the device driver while the disk sectors are being transferred. For instance, the `nr_sectors` field stores the number of sectors yet to be transferred in the whole request, while the `current_nr_sectors` field stores the number of sectors yet to be transferred in the current bio.

The `flags` field stores a large number of flags, which are listed in Table 14-8. The most important one is, by far, `REQ_RW`, which determines the direction of the data transfer.

Table 14-8. The flags of the request descriptor

Flag	Description
REQ_RW	Direction of data transfer: READ (0) or WRITE (1)
REQ_FAILFAST	Requests says to not retry the I/O operation in case of error
REQ_SOFTBARRIER	Request acts as a barrier for the I/O scheduler
REQ_HARDBARRIER	Request acts as a barrier for the I/O scheduler and the device driver—it should be processed after older requests and before newer ones
REQ_CMD	Request includes a normal read or write I/O data transfer
REQ_NOMERGE	Request should not be extended or merged with other requests
REQ_STARTED	Request is being processed
REQ_DONTPREP	Do not invoke the `prep_rq_fn` request queue's method to prepare in advance the commands to be sent to the hardware device
REQ_QUEUED	Request is tagged—that is, it refers to a hardware device that can manage many outstanding data transfers at the same time
REQ_PC	Request includes a direct command to be sent to the hardware device
REQ_BLOCK_PC	Same as previous flag, but the command is included in a bio
REQ_SENSE	Request includes a "sense" request command (for SCSI and ATAPI devices)
REQ_FAILED	Set when a sense or direct command in the request did not work as expected
REQ_QUIET	Request says to not generate kernel messages in case of I/O errors
REQ_SPECIAL	Request includes a special command for the hardware device (e.g., drive reset)

Table 14-8. The flags of the request descriptor (continued)

Flag	Description
REQ_DRIVE_CMD	Request includes a special command for IDE disks
REQ_DRIVE_TASK	Request includes a special command for IDE disks
REQ_DRIVE_TASKFILE	Request includes a special command for IDE disks
REQ_PREEMPT	Request replaces the current request in front of the queue (only for IDE disks)
REQ_PM_SUSPEND	Request includes a power-management command to suspend the hardware device
REQ_PM_RESUME	Request includes a power-management command to awaken the hardware device
REQ_PM_SHUTDOWN	Request includes a power-management command to switch off the hardware device
REQ_BAR_PREFLUSH	Request includes a "flush queue" command to be sent to the disk controller
REQ_BAR_POSTFLUSH	Request includes a "flush queue" command, which has been sent to the disk controller

Managing the allocation of request descriptors

The limited amount of free dynamic memory may become, under very heavy loads and high disk activity, a bottleneck for processes that want to add a new request into a request queue q. To cope with this kind of situation, each request_queue descriptor includes a request_list data structure, which consists of:

- A pointer to a memory pool of request descriptors (see the section "Memory Pools" in Chapter 8).
- Two counters for the number of requests descriptors allocated for READ and WRITE requests, respectively.
- Two flags indicating whether a recent allocation for a READ or WRITE request, respectively, failed.
- Two wait queues storing the processes sleeping for available READ and WRITE request descriptors, respectively.
- A wait queue for the processes waiting for a request queue to be flushed (emptied).

The blk_get_request() function tries to get a free request descriptor from the memory pool of a given request queue; if memory is scarce and the memory pool is exhausted, the function either puts the current process to sleep or—if the kernel control path cannot block—returns NULL. If the allocation succeeds, the function stores in the rl field of the request descriptor the address of the request_list data structure of the request queue. The blk_put_request() function releases a request descriptor; if its reference counter becomes zero, the descriptor is given back to the memory pool from which it was taken.

Avoiding request queue congestion

Each request queue has a maximum number of allowed pending requests. The nr_requests field of the request queue descriptor stores the maximum number of allowed pending requests for each data transfer direction. By default, a queue has at most 128

pending read requests and 128 pending write requests. If the number of pending read (write) requests exceeds `nr_requests`, the queue is marked as full by setting the `QUEUE_FLAG_READFULL` (`QUEUE_FLAG_WRITEFULL`) flag in the `queue_flags` field of the request queue descriptor, and blockable processes trying to add requests for that data transfer direction are put to sleep in the corresponding wait queue of the `request_list` data structure.

A filled-up request queue impacts negatively on the system's performance, because it forces many processes to sleep while waiting for the completion of I/O data transfers. Thus, if the number of pending requests for a given direction exceeds the value stored in the `nr_congestion_on` field of the request descriptor (by default, 113), the kernel regards the queue as *congested* and tries to slow down the creation rate of the new requests. A congested request queue becomes uncongested when the number of pending requests falls below the value of the `nr_congestion_off` field (by default, 111). The `blk_congestion_wait()` function puts the current process to sleep until any request queue becomes uncongested or a time-out elapses.

Activating the Block Device Driver

As we saw earlier, it's expedient to delay activation of the block device driver in order to increase the chances of clustering requests for adjacent blocks. The delay is accomplished through a technique known as *device plugging* and *unplugging*.[*] As long as a block device driver is plugged, the device driver is not activated even if there are requests to be processed in the driver's queues.

The `blk_plug_device()` function plugs a block device—or more precisely, a request queue serviced by some block device driver. Essentially, the function receives as an argument the address q of a request queue descriptor. It sets the `QUEUE_FLAG_PLUGGED` bit in the `q->queue_flags` field; then, it restarts the dynamic timer embedded in the `q->unplug_timer` field.

The `blk_remove_plug()` function unplugs a request queue q: it clears the `QUEUE_FLAG_PLUGGED` flag and cancels the execution of the `q->unplug_timer` dynamic timer. This function can be explicitly invoked by the kernel when all mergeable requests "in sight" have been added to the queue. Moreover, the I/O scheduler unplugs a request queue if the number of pending requests in the queue exceeds the value stored in the `unplug_thres` field of the request queue descriptor (by default, 4).

If a device remains plugged for a time interval of length `q->unplug_delay` (usually 3 milliseconds), the dynamic timer activated by `blk_plug_device()` elapses, thus the `blk_unplug_timeout()` function is executed. As a consequence, the *kblockd* kernel thread servicing the `kblockd_workqueue` work queue is awakened (see the section

---

[*] If you are confused by the terms "plugging" and "unplugging," you might consider them equivalent to "deactivating" and "activating," respectively.

"Work Queues" in Chapter 4). This kernel thread executes the function whose address is stored in the q->unplug_work data structure—that is, the blk_unplug_work() function. In turn, this function invokes the q->unplug_fn method of the request queue, which is usually implemented by the generic_unplug_device() function. The generic_unplug_device() function takes care of unplugging the block device: first, it checks whether the queue is still active; then, it invokes blk_remove_plug(); and finally, it executes the strategy routine—request_fn method—to start processing the next request in the queue (see the section "Device Driver Registration and Initialization" later in this chapter).

I/O Scheduling Algorithms

When a new request is added to a request queue, the generic block layer invokes the I/O scheduler to determine that exact position of the new element in the queue. The I/O scheduler tries to keep the request queue sorted sector by sector. If the requests to be processed are taken sequentially from the list, the amount of disk seeking is significantly reduced because the disk head moves in a linear way from the inner track to the outer one (or vice versa) instead of jumping randomly from one track to another. This heuristic is reminiscent of the algorithm used by elevators when dealing with requests coming from different floors to go up or down. The elevator moves in one direction; when the last booked floor is reached in one direction, the elevator changes direction and starts moving in the other direction. For this reason, I/O schedulers are also called *elevators*.

Under heavy load, an I/O scheduling algorithm that strictly follows the order of the sector numbers is not going to work well. In this case, indeed, the completion time of a data transfer strongly depends on the physical position of the data on the disk. Thus, if a device driver is processing requests near the top of the queue (lower sector numbers), and new requests with low sector numbers are continuously added to the queue, then the requests in the tail of the queue can easily starve. I/O scheduling algorithms are thus quite sophisticated.

Currently, Linux 2.6 offers four different types of I/O schedulers—or elevators—called "Anticipatory," "Deadline," "CFQ (Complete Fairness Queueing)," and "Noop (No Operation)." The default elevator used by the kernel for most block devices is specified at boot time with the kernel parameter elevator=*<name>*, where *<name>* is one of the following: as, deadline, cfq, and noop. If no boot time argument is given, the kernel uses the "Anticipatory" I/O scheduler. Anyway, a device driver can replace the default elevator with another one; a device driver can also define its custom I/O scheduling algorithm, but this is very seldom done.

Furthermore, the system administrator can change at runtime the I/O scheduler for a specific block device. For instance, to change the I/O scheduler used in the master disk of the first IDE channel, the administrator can write an elevator name into the */sys/block/hda/queue/scheduler* file of the *sysfs* special filesystem (see the section "The sysfs Filesystem" in Chapter 13).

The I/O scheduler algorithm used in a request queue is represented by an *elevator object* of type `elevator_t`; its address is stored in the `elevator` field of the request queue descriptor. The elevator object includes several methods covering all possible operations of the elevator: linking and unlinking the elevator to a request queue, adding and merging requests to the queue, removing requests from the queue, getting the next request to be processed from the queue, and so on. The elevator object also stores the address of a table including all information required to handle the request queue. Furthermore, each request descriptor includes an `elevator_private` field that points to an additional data structure used by the I/O scheduler to handle the request.

Let us now briefly describe the four I/O scheduling algorithms, from the simplest one to the most sophisticated one. Be warned that designing an I/O scheduler is much like designing a CPU scheduler (see Chapter 7): the heuristics and the values of the adopted constants are the result of an extensive amount of testing and benchmarking.

Generally speaking, all algorithms make use of a *dispatch queue*, which includes all requests sorted according to the order in which the requests should be processed by the device driver—the next request to be serviced by the device driver is always the first element in the dispatch queue. The dispatch queue is actually the request queue rooted at the `queue_head` field of the request queue descriptor. Almost all algorithms also make use of additional queues to classify and sort requests. All of them allow the device driver to add bios to existing requests and, if necessary, to merge two "adjacent" requests.

The "Noop" elevator

This is the simplest I/O scheduling algorithm. There is no ordered queue: new requests are always added either at the front or at the tail of the dispatch queue, and the next request to be processed is always the first request in the queue.

The "CFQ" elevator

The main goal of the "Complete Fairness Queueing" elevator is ensuring a fair allocation of the disk I/O bandwidth among all the processes that trigger the I/O requests. To achieve this result, the elevator makes use of a large number of sorted queues—by default, 64—that store the requests coming from the different processes. Whenever a requested is handed to the elevator, the kernel invokes a hash function that converts the thread group identifier of the current process (usually it corresponds to the PID, see the section "Identifying a Process" in Chapter 3) into the index of a queue; then, the elevator inserts the new request at the tail of this queue. Therefore, requests coming from the same process are always inserted in the same queue.

To refill the dispatch queue, the elevator essentially scans the I/O input queues in a round-robin fashion, selects the first nonempty queue, and moves a batch of requests from that queue into the tail of the dispatch queue.

The "Deadline" elevator

Besides the dispatch queue, the "Deadline" elevator makes use of four queues. Two of them—the *sorted queues*—include the read and write requests, respectively, ordered according to their initial sector numbers. The other two—the *deadline queues*—include the same read and write requests sorted according to their "deadlines." These queues are introduced to avoid *request starvation*, which occurs when the elevator policy ignores for a very long time a request because it prefers to handle other requests that are closer to the last served one. A request *deadline* is essentially an expire timer that starts ticking when the request is passed to the elevator. By default, the expire time of read requests is 500 milliseconds, while the expire time for write requests is 5 seconds—read requests are privileged over write requests because they usually block the processes that issued them. The deadline ensures that the scheduler looks at a request if it's been waiting a long time, even if it is low in the sort.

When the elevator must replenish the dispatch queue, it first determines the data direction of the next request. If there are both read and write requests to be dispatched, the elevator chooses the "read" direction, unless the "write" direction has been discarded too many times (to avoid write requests starvation).

Next, the elevator checks the deadline queue relative to the chosen direction: if the deadline of the first request in the queue is elapsed, the elevator moves that request to the tail of the dispatch queue; it also moves a batch of requests taken from the sorted queue, starting from the request following the expired one. The length of this batch is longer if the requests happen to be physically adjacent on disks, shorter otherwise.

Finally, if no request is expired, the elevator dispatches a batch of requests starting with the request following the last one taken from the sorted queue. When the cursor reaches the tail of the sorted queue, the search starts again from the top ("one-way elevator").

The "Anticipatory" elevator

The "Anticipatory" elevator is the most sophisticated I/O scheduler algorithm offered by Linux. Basically, it is an evolution of the "Deadline" elevator, from which it borrows the fundamental mechanism: there are two deadline queues and two sorted queues; the I/O scheduler keeps scanning the sorted queues, alternating between read and write requests, but giving preference to the read ones. The scanning is basically sequential, unless a request expires. The default expire time for read requests is 125 milliseconds, while the default expire time for write requests is 250 milliseconds. The elevator, however, follows some additional heuristics:

- In some cases, the elevator might choose a request behind the current position in the sorted queue, thus forcing a backward seek of the disk head. This happens, typically, when the seek distance for the request behind is less than half the seek distance of the request after the current position in the sorted queue.

- The elevator collects statistics about the patterns of I/O operations triggered by every process in the system. Right after dispatching a read request that comes from some process P, the elevator checks whether the next request in the sorted queue comes from the same process P. If so, the next request is dispatched immediately. Otherwise, the elevator looks at the collected statistics about process P: if it decides that process P will likely issue another read request soon, then it stalls for a short period of time (by default, roughly 7 milliseconds). Thus, the elevator might anticipate a read request coming from process P that is "close" on disk to the request just dispatched.

Issuing a Request to the I/O Scheduler

As seen in the section "Submitting a Request" earlier in this chapter, the generic_ make_request() function invokes the make_request_fn method of the request queue descriptor to transmit a request to the I/O scheduler. This method is usually implemented by the __make_request() function; it receives as its parameters a request_ queue descriptor q and a bio descriptor bio, and it performs the following operations:

1. Invokes the blk_queue_bounce() function to set up a bounce buffer, if required (see later). If a bounce buffer was created, the __make_request() function operates on it rather than on the original bio.

2. Invokes the I/O scheduler function elv_queue_empty() to check whether there are pending requests in the request queue—notice that the dispatch queue might be empty, but other queues of the I/O scheduler might contain pending requests. If there are no pending requests, it invokes the blk_plug_device() function to plug the request queue (see the section "Activating the Block Device Driver" earlier in this chapter), and jumps to step 5.

3. Here the request queue includes pending requests. Invokes the elv_merge() I/O scheduler function to check whether the new bio can be merged inside an existing request. The function may return three possible values:

   - ELEVATOR_NO_MERGE: the bio cannot be included in an already existing request: in that case, the function jumps to step 5.

   - ELEVATOR_BACK_MERGE: the bio might be added as the last bio of some request req: in that case, the function invokes the q->back_merge_fn method to check whether the request can be extended. If not, the function jumps to step 5. Otherwise it inserts the bio descriptor at the tail of the req's list and updates the req's fields. Then, it tries to merge the request with a following request (the new bio might fill a hole between the two requests).

- ELEVATOR_FRONT_MERGE: the bio can be added as the first bio of some request req: in that case, the function invokes the q->front_merge_fn method to check whether the request can be extended. If not, it jumps to step 5. Otherwise, it inserts the bio descriptor at the head of the req's list and updates the req's fields. Then, the function tries to merge the request with the preceding request.

4. The bio has been merged inside an already existing request. Jumps to step 7 to terminate the function.

5. Here the bio must be inserted in a new request. Allocates a new request descriptor. If there is no free memory, the function suspends the current process, unless the BIO_RW_AHEAD flag in bio->bi_rw is set, which means that the I/O operation is a read-ahead (see Chapter 16); in this case, the function invokes bio_endio() and terminates: the data transfer will not be executed. For a description of bio_endio(), see step 1 of generic_make_request() in the earlier section "Submitting a Request."

6. Initializes the fields of the request descriptor. In particular:

   a. Initializes the various fields that store the sector numbers, the current bio, and the current segment according to the contents of the bio descriptor.

   b. Sets the REQ_CMD flag in the flags field (this is a normal read or write operation).

   c. If the page frame of the first bio segment is in low memory, it sets the buffer field to the linear address of that buffer.

   d. Sets the rq_disk field with the bio->bi_bdev->bd_disk address.

   e. Inserts the bio in the request list.

   f. Sets the start_time field to the value of jiffies.

7. All done. Before terminating, however, it checks whether the BIO_RW_SYNC flag in bio->bi_rw is set. If so, it invokes generic_unplug_device() on the request queue to unplug the driver (see the section "Activating the Block Device Driver" earlier in this chapter).

8. Terminates.

If the request queue was not empty before invoking __make_request(), either the request queue is already unplugged, or it will be unplugged soon—because each plugged request queue q with pending requests has a running q->unplug_timer dynamic timer. On the other hand, if the request queue was empty, the __make_request() function plugs it. Sooner (on exiting from __make_request(), if the BIO_RW_SYNC bio flag is set) or later (in the worst case, when the unplug timer decays), the request queue will be unplugged. In any case, eventually the strategy routine of the block device driver will take care of the requests in the dispatch queue (see the section "Device Driver Registration and Initialization" earlier in this chapter).

The blk_queue_bounce( ) function

The `blk_queue_bounce()` function looks at the flags in `q->bounce_gfp` and at the threshold in `q->bounce_pfn` to determine whether *buffer bouncing* might be required. This happens when some of the buffers in the request are located in high memory and the hardware device is not able to address them.

Older DMA for ISA buses only handled 24-bit physical addresses. In this case, the buffer bouncing threshold is set to 16 MB, that is, to page frame number 4096. Block device drivers, however, do not usually rely on buffer bouncing when dealing with older devices; rather, they prefer to directly allocate the DMA buffers in the `ZONE_DMA` memory zone.

If the hardware device cannot cope with buffers in high memory, the function checks whether some of the buffers in the bio must really be bounced. In this case, it makes a copy of the bio descriptor, thus creating a *bounce bio*; then, for each segment's page frame having number equal to or greater than `q->bounce_pfn`, it performs the following steps:

1. Allocates a page frame in the `ZONE_NORMAL` or `ZONE_DMA` memory zone, according to the allocation flags.
2. Updates the `bv_page` field of the segment in the bounce bio so that it points to the descriptor of the new page frame.
3. If `bio->bio_rw` specifies a write operation, it invokes `kmap()` to temporarily map the high memory page in the kernel address space, copies the high memory page onto the low memory page, and invokes `kunmap()` to release the mapping.

The `blk_queue_bounce()` function then sets the `BIO_BOUNCED` flag in the bounce bio, initializes a specific `bi_end_io` method for the bounce bio, and finally stores in the `bi_private` field of the bounce bio the pointer to the original bio. When the I/O data transfer on the bounce bio terminates, the function that implements the `bi_end_io` method copies the data to the high memory buffer (only for a read operation) and releases the bounce bio.

Block Device Drivers

Block device drivers are the lowest component of the Linux block subsystem. They get requests from I/O scheduler, and do whatever is required to process them.

Block device drivers are, of course, integrated within the device driver model described in the section "The Device Driver Model" in Chapter 13. Therefore, each of them refers to a `device_driver` descriptor; moreover, each disk handled by the driver is associated with a `device` descriptor. These descriptors, however, are rather generic: the block I/O subsystem must store additional information for each block device in the system.

Block Devices

A block device driver may handle several block devices. For instance, the IDE device driver can handle several IDE disks, each of which is a separate block device. Furthermore, each disk is usually partitioned, and each partition can be seen as a logical block device. Clearly, the block device driver must take care of all VFS system calls issued on the block device files associated with the corresponding block devices.

Each block device is represented by a `block_device` descriptor, whose fields are listed in Table 14-9.

Table 14-9. The fields of the block device descriptor

Type	Field	Description
dev_t	bd_dev	Major and minor numbers of the block device
struct inode *	bd_inode	Pointer to the inode of the file associated with the block device in the *bdev* filesystem
int	bd_openers	Counter of how many times the block device has been opened
struct semaphore	bd_sem	Semaphore protecting the opening and closing of the block device
struct semaphore	bd_mount_sem	Semaphore used to forbid new mounts on the block device
struct list_head	bd_inodes	Head of a list of inodes of opened block device files for this block device
void *	bd_holder	Current holder of block device descriptor
int	bd_holders	Counter for multiple settings of the bd_holder field
struct block_device *	bd_contains	If block device is a partition, it points to the block device descriptor of the whole disk; otherwise, it points to this block device descriptor
unsigned	bd_block_size	Block size
struct hd_struct *	bd_part	Pointer to partition descriptor (NULL if this block device is not a partition)
unsigned	bd_part_count	Counter of how many times partitions included in this block device have been opened
int	bd_invalidated	Flag set when the partition table on this block device needs to be read
struct gendisk *	bd_disk	Pointer to gendisk structure of the disk underlying this block device
struct list_head *	bd_list	Pointers for the block device descriptor list
struct backing_dev_info *	bd_inode_backing_dev_info	Pointer to a specialized backing_dev_info descriptor for this block device (usually NULL)
unsigned long	bd_private	Pointer to private data of the block device holder

All block device descriptors are inserted in a global list, whose head is represented by the `all_bdevs` variable; the pointers for list linkage are in the `bd_list` field of the block device descriptor.

If the block device descriptor refers to a disk partition, the `bd_contains` field points to the descriptor of the block device associated with the whole disk, while the `bd_part` field points to the `hd_struct` partition descriptor (see the section "Representing Disks and Disk Partitions" earlier in this chapter). Otherwise, if the block device descriptor refers to a whole disk, the `bd_contains` field points to the block device descriptor itself, and the `bd_part_count` field records how many times the partitions on the disk have been opened.

The `bd_holder` field stores a linear address representing the *holder* of the block device. The holder is not the block device driver that services the I/O data transfers of the device; rather, it is a kernel component that makes use of the device and has exclusive, special privileges (for instance, it can freely use the `bd_private` field of the block device descriptor). Typically, the holder of a block device is the filesystem mounted over it. Another common case occurs when a block device file is opened for exclusive access: the holder is the corresponding file object.

The `bd_claim()` function sets the `bd_holder` field with a specified address; conversely, the `bd_release()` function resets the field to `NULL`. Be aware, however, that the same kernel component can invoke `bd_claim()` many times; each invocation increases the `bd_holders` field. To release the block device, the kernel component must invoke `bd_release()` a corresponding number of times.

Figure 14-3 refers to a whole disk and illustrates how the block device descriptors are linked to the other main data structures of the block I/O subsystem.
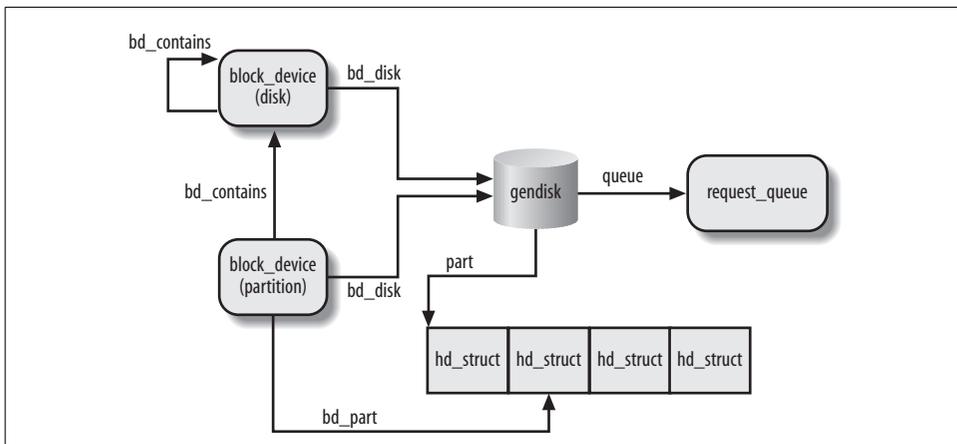
Figure 14-3. Linking the block device descriptors with the other structures of the block subsystem

Accessing a block device

When the kernel receives a request for opening a block device file, it must first determine whether the device file is already open. In fact, if the file is already open, the kernel must not create and initialize a new block device descriptor; rather, it should update the already existing block device descriptor. To complicate life, block device files that have the same major and minor numbers but different pathnames are viewed by the VFS as different files, although they really refer to the same block device. Therefore, the kernel cannot determine whether a block device is already in use by simply checking for the existence in the inode cache of an object for a block device file.

The relationship between a major and minor number and the corresponding block device descriptor is maintained through the *bdev* special filesystem (see the section "Special Filesystems" in Chapter 12). Each block device descriptor is coupled with a *bdev* special file: the `bd_inode` field of the block device descriptor points to the corresponding *bdev* inode; conversely, such an inode encodes both the major and minor numbers of the block device and the address of the corresponding descriptor.

The `bdget()` function receives as its parameter the major and minor numbers of a block device: It looks up in the *bdev* filesystem the associated inode; if such inode does not exist, the function allocates a new inode and new block device descriptor. In any case, the function returns the address of the block device descriptor corresponding to given major and minor numbers.

Once the block device descriptor for a block device has been found, the kernel can determine whether the block device is currently in use by checking the value of the `bd_openers` field: if it is positive, the block device is already in use (possibly by means of a different device file). The kernel also maintains a list of inode objects relative to opened block device files. This list is rooted at the `bd_inodes` field of the block device descriptor; the `i_devices` field of the inode object stores the pointers for the previous and next element in this list.

Device Driver Registration and Initialization

Let's now explain the essential steps involved in setting up a new device driver for a block device. Clearly, the description that follows is very succinct, nevertheless it could be useful to understand how and when the main data structures used by the block I/O subsystem are initialized.

We silently omit many steps required for all kinds of device drivers and already mentioned in Chapter 13. For example, we skip all steps required for registering the driver itself (see the section "The Device Driver Model" in Chapter 13). Usually, the block device belongs to a standard bus architecture such as PCI or SCSI, and the kernel offers helper functions that, as a side effect, register the driver in the device driver model.

Defining a custom driver descriptor

First of all, the device driver needs a custom descriptor *foo* of type *foo_dev_t* holding the data required to drive the hardware device. For every device, the descriptor will store information such as the I/O ports used to program the device, the IRQ line of the interrupts raised by the device, the internal status of the device, and so on. The descriptor must also include a few fields required by the block I/O subsystem:

```
struct foo_dev_t {
    [...]
    spinlock_t lock;
    struct gendisk *gd;
    [...]
} foo;
```

The `lock` field is a spin lock used to protect the fields of the *foo* descriptor; its address is often passed to kernel helper functions, which can thus protect the data structures of the block I/O subsystem specific to the driver. The `gd` field is a pointer to the `gendisk` descriptor that represents the whole block device (disk) handled by this driver.

Reserving the major number

The device driver must reserve a major number for its own purposes. Traditionally, this is done by invoking the `register_blkdev()` function:

```
err = register_blkdev(FOO_MAJOR, "foo");
if (err) goto error_major_is_busy;
```

This function is very similar to `register_chrdev()` presented in the section "Assigning Device Numbers" in Chapter 13: it reserves the major number *FOO_MAJOR* and associates the name *foo* to it. Notice that there is no way to allocate a subrange of minor numbers, because there is no analog of `register_chrdev_region()`; moreover, no link is established between the reserved major number and the data structures of the driver. The only visible effect of `register_blkdev()` is to include a new item in the list of registered major numbers in the */proc/devices* special file.

Initializing the custom descriptor

All the fields of the *foo* descriptor must be initialized properly before making use of the driver. To initialize the fields related to the block I/O subsystem, the device driver could execute the following instructions:

```
spin_lock_init(&foo.lock);
foo.gd = alloc_disk(16);
if (!foo.gd) goto error_no_gendisk;
```

The driver initializes the spin lock, then allocates the disk descriptor. As shown earlier in Figure 14-3, the `gendisk` structure is crucial for the block I/O subsystem, because it references many other data structures. The `alloc_disk()` function allocates also the array that stores the partition descriptors of the disk. The argument of

the function is the number of `hd_struct` elements in the array; the value 16 means that the driver can support disks containing up to 15 partitions (partition 0 is not used).

Initializing the gendisk descriptor

Next, the driver initializes some fields of the `gendisk` descriptor:

```
foo.gd->private_data = &foo;
foo.gd->major = FOO_MAJOR;
foo.gd->first_minor = 0;
foo.gd->minors = 16;
set_capacity(foo.gd, foo_disk_capacity_in_sectors);
strcpy(foo.gd->disk_name, "foo");
foo.gd->fops = &foo_ops;
```

The address of the *foo* descriptor is saved in the `private_data` of the `gendisk` structure, so that low-level driver functions invoked as methods by the block I/O subsystem can quickly find the driver descriptor—this improves efficiency if the driver can handle more than one disk at a time. The `set_capacity()` function initializes the `capacity` field with the size of the disk in 512-byte sectors—this value is likely determined by probing the hardware and asking about the disk parameters.

Initializing the table of block device methods

The `fops` field of the `gendisk` descriptor is initialized with the address of a custom table of block device methods (see Table 14-4 earlier in this chapter).[*] Quite likely, the *foo_ops* table of the device driver includes functions specific to the device driver. As an example, if the hardware device supports removable disks, the generic block layer may invoke the `media_changed` method to check whether the disk is changed since the last mount or open operation on the block device. This check is usually done by sending some low-level commands to the hardware controller, thus the implementation of the `media_changed` method is always specific to the device driver.

Similarly, the `ioctl` method is only invoked when the generic block layer does not know how to handle some *ioctl* command. For instance, the method is typically invoked when an `ioctl()` system call asks about the *disk geometry*, that is, the number of cylinders, tracks, sectors, and heads used by the disk. Thus, the implementation of this method is specific to the device driver.

Allocating and initializing a request queue

Our brave device driver designer might now set up a request queue that will collect the requests waiting to be serviced. This can be easily done as follows:

---

[*] The block device methods should not be confused with the block device file operations; see the section "Opening a Block Device File" later in this chapter.

```
foo.gd->rq = blk_init_queue(foo_strategy, &foo.lock);
if (!foo.gd->rq) goto error_no_request_queue;
blk_queue_hardsect_size(foo.gd->rd, foo_hard_sector_size);
blk_queue_max_sectors(foo.gd->rd, foo_max_sectors);
blk_queue_max_hw_segments(foo.gd->rd, foo_max_hw_segments);
blk_queue_max_phys_segments(foo.gd->rd, foo_max_phys_segments);
```

The `blk_init_queue()` function allocates a request queue descriptor and initializes many of its fields with default values. It receives as its parameters the address of the device descriptor's spin lock—for the `foo.gd->rq->queue_lock` field—and the address of the strategy routine of the device driver—for the `foo.gd->rq->request_fn` field; see the next section; "The Strategy Routine." The `blk_init_queue()` function also initializes the `foo.gd->rq->elevator` field, forcing the driver to use the default I/O scheduler algorithm. If the device driver wants to use a different elevator, it may later override the address in the `elevator` field.

Next, some helper functions set various fields of the request queue descriptor with the proper values for the device driver (look at Table 14-6 for the similarly named fields).

Setting up the interrupt handler

As described in the section "I/O Interrupt Handling" in Chapter 4, the driver needs to register the IRQ line for the device. This can be done as follows:

```
request_irq(foo_irq, foo_interrupt,
            SA_INTERRUPT|SA_SHIRQ, "foo", NULL);
```

The `foo_interrupt()` function is the interrupt handler for the device; we discuss some of its peculiarities in the section "The Interrupt Handler" later in this chapter).

Registering the disk

Finally, all the device driver's data structures are ready: the last step of the initialization phase consists of "registering" and activating the disk. This can be achieved simply by executing:

```
add_disk(foo.gd);
```

The `add_disk()` function receives as its parameter the address of the `gendisk` descriptor, and essentially executes the following operations:

1. Sets the `GENHD_FL_UP` flag of `gd->flags`.
2. Invokes `kobj_map()` to establish the link between the device driver and the device's major number with its associated range of minor numbers (see the section "Character Device Drivers" in Chapter 13; be warned that in this case the kobject mapping domain is represented by the `bdev_map` variable).
3. Registers the kobject included in the `gendisk` descriptor in the device driver model as a new device serviced by the device driver (e.g., */sys/block/foo*).

4. Scans the partition table included in the disk, if any; for each partition found, properly initializes the corresponding hd_struct descriptor in the *foo*.gd->part array. Also registers the partitions in the device driver model (e.g., */sys/block/foo/foo1*).

5. Registers the kobject embedded in the request queue descriptor in the device driver model (e.g., */sys/block/foo/queue*).

Once add_disk() returns, the device driver is actively working. The function that carried on the initialization phase terminates; the strategy routine and the interrupt handler take care of each request passed to the device driver by the I/O scheduler.

The Strategy Routine

The strategy routine is a function—or a group of functions—of the block device driver that interacts with the hardware block device to satisfy the requests collected in the dispatch queue. The strategy routine is invoked by means of the request_fn method of the request queue descriptor—the *foo*_strategy() function in the example carried on in the previous section. The I/O scheduler layer passes to this function the address q of the request queue descriptor.

As we'll see, the strategy routine is usually started after inserting a new request in an empty request queue. Once activated, the block device driver should handle all requests in the queue and terminate when the queue is empty.

A naïve implementation of the strategy routine could be the following: for each element in the dispatch queue, remove it from the queue, interact with the block device controller to service the request, and wait until the data transfer completes. Then proceed with the next request in the dispatch queue.

Such an implementation is not very efficient. Even assuming that the data can be transferred using DMA, the strategy routine must suspend itself while waiting for I/O completion. This means that the strategy routine should execute on a dedicated kernel thread (we don't want to penalize an unrelated user process, do we?). Moreover, such a driver would not be able to support modern disk controllers that can process multiple I/O data transfers at a time.

Therefore, most block device drivers adopt the following strategy:

• The strategy routine starts a data transfer for the first request in the queue and sets up the block device controller so that it raises an interrupt when the data transfer completes. Then the strategy routine terminates.

• When the disk controller raises the interrupt, the interrupt handler invokes the strategy routine again (often directly, sometimes by activating a work queue). The strategy routine either starts another data transfer for the current request or, if all the chunks of data of the request have been transferred, removes the request from the dispatch queue and starts processing the next request.

Requests can be composed of several bios, which in turn can be composed of several segments. Basically, block device drivers make use of DMA in two ways:

- The driver sets up a different DMA transfer to service each segment in each bio of the request
- The driver sets up a single scatter-gather DMA transfer to service all segments in all bios of the request

Ultimately, the design of the strategy routine of the device drivers depends on the characteristics of the block controller. Each physical block device is inherently different from all others (for example, a floppy driver groups blocks in disk tracks and transfers a whole track in a single I/O operation), so making general assumptions on how a device driver should service a request is meaningless.

In our example, the *foo*_strategy() strategy routine could execute the following actions:

1. Gets the current request from the dispatch queue by invoking the elv_next_ request() I/O scheduler helper function. If the dispatch queue is empty, the strategy routine returns:

    ```
    req = elv_next_request(q);
    if (!req) return;
    ```

2. Executes the blk_fs_request macro to check whether the REQ_CMD flag of the request is set, that is, whether the request contains a normal read or write operation:

    ```
    if (!blk_fs_request(req))
        goto handle_special_request;
    ```

3. If the block device controller supports scatter-gather DMA, it programs the disk controller so as to perform the data transfer for the whole request and to raise an interrupt when the transfer completes. The blk_rq_map_sg() helper function returns a scatter-gather list that can be immediately used to start the transfer.

4. Otherwise, the device driver must transfer the data segment by segment. In this case, the strategy routine executes the rq_for_each_bio and bio_for_each_ segment macros, which walk the list of bios and the list of segments inside each bio, respectively:

    ```
    rq_for_each_bio(bio, rq)
        bio_for_each_segment(bvec, bio, i) {
            /* transfer the i-th segment bvec */
            local_irq_save(flags);
            addr = kmap_atomic(bvec->bv_page, KM_BIO_SRC_IRQ);
            foo_start_dma_transfer(addr+bvec->bv_offset, bvec->bv_len);
            kunmap_atomic(bvec->bv_page, KM_BIO_SRC_IRQ);
            local_irq_restore(flags);
        }
    ```

    The kmap_atomic() and kunmap_atomic() functions are required if the data to be transferred can be in high memory. The *foo*_start_dma_transfer() function

programs the hardware device so as to start the DMA transfer and to raise an interrupt when the I/O operation completes.

5. Returns.

The Interrupt Handler

The interrupt handler of a block device driver is activated when a DMA transfer terminates. It should check whether all chunks of data in the request have been transferred. If so, the interrupt handler invokes the strategy routine to process the next request in the dispatch queue. Otherwise, the interrupt handler updates the field of the request descriptor and invokes the strategy routine to process the data transfer yet to be performed.

A typical fragment of the interrupt handler of our *foo* device driver is the following:

```
irqreturn_t foo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct foo_dev_t *p = (struct foo_dev_t *) dev_id;
    struct request_queue *rq = p->gd->rq;
        [...]
    if (!end_that_request_first(rq, uptodate, nr_sectors)) {
        blkdev_dequeue_request(rq);
        end_that_request_last(rq);
    }
    rq->request_fn(rq);
        [...]
    return IRQ_HANDLED;
}
```

The job of ending a request is split in two functions called end_that_request_first() and end_that_request_last().

The end_that_request_first() function receives as arguments a request descriptor, a flag indicating if the DMA data transfer completed successfully, and the number of sectors transferred in the DMA transfer (the end_that_request_chunk() function is similar, but it receives the number of bytes transferred instead of the number of sectors). Essentially, the function scans the bios in the request and the segments inside each bio, then updates the fields of the request descriptor in such a way to:

* Set the bio field so that it points to the first unfinished bio in the request.
* Set the bi_idx of the unfinished bio so that it points to the first unfinished segment.
* Set the bv_offset and bv_len fields of the unfinished segment so that they specify the data yet to be transferred.

The function also invokes bio_endio() on each bio that has been completely transferred.

The `end_that_request_first()` function returns 0 if all chunks of data in the request have been transferred; otherwise, it returns 1. If the returned value is 1, the interrupt handler restarts the strategy routine, which thus continues processing the same request. Otherwise, the interrupt handler removes the request from the request queue (typically by using `blkdev_dequeue_request()`), invokes the `end_that_request_last()` helper function, and restarts the strategy routine to process the next request in the dispatch queue.

The `end_that_request_last()` function updates some disk usage statistics, removes the request descriptor from the dispatch queue of the `rq->elevator` I/O scheduler, wakes up any process sleeping in the `waiting` completion of the request descriptor, and releases that descriptor.

Opening a Block Device File

We conclude this chapter by describing the steps performed by the VFS when opening a block device file.

The kernel opens a block device file every time that a filesystem is mounted over a disk or partition, every time that a swap partition is activated, and every time that a User Mode process issues an `open()` system call on a block device file. In all cases, the kernel executes essentially the same operations: it looks for the block device descriptor (possibly allocating a new descriptor if the block device is not already in use), and sets up the file operation methods for the forthcoming data transfers.

In the section "VFS Handling of Device Files" in Chapter 13, we described how the `dentry_open()` function customizes the methods of the file object when a device file is opened. In this case, the `f_op` field of the file object is set to the address of the `def_blk_fops` table, whose content is shown in Table 14-10.

Table 14-10. The default block device file operations (def_blk_fops table)

Method	Function
open	blkdev_open()
release	blkdev_close()
llseek	block_llseek()
read	generic_file_read()
write	blkdev_file_write()
aio_read	generic_file_aio_read()
aio_write	blkdev_file_aio_write()
mmap	generic_file_mmap()
fsync	block_fsync()
ioctl	block_ioctl()
compat-ioctl	compat_blkdev_ioctl()

Method	Function
readv	generic_file_readv()
writev	generic_file_write_nolock()
sendfile	generic_file_sendfile()

Here we are only concerned with the open method, which is invoked by the dentry_open() function. The blkdev_open() function receives as its parameters inode and filp, which store the addresses of the inode and file objects respectively; the function essentially executes the following steps:

1. Executes bd_acquire(inode) to get the address bdev of the block device descriptor. In turn, this function receives the inode object address and performs the following steps:

    a. Checks whether the inode->i_bdev field of the inode object is not NULL; if it is, the block device file has been opened already, and this field stores the address of the corresponding block descriptor. In this case, the function increases the usage counter of the inode->i_bdev->bd_inode inode of the *bdev* special filesystem associated with the block device, and returns the address inode->i_bdev of the descriptor.

    b. Here the block device file has not been opened yet. Executes bdget(inode->i_rdev) to get the address of the block device descriptor corresponding to the major and minor number of the block device file (see the section "Block Devices" earlier in this chapter). If the descriptor does not already exist, bdget() allocates it; notice however that the descriptor might already exist, for instance because the block device is already being accessed by means of another block device file.

    c. Stores the block device descriptor address in inode->i_bdev, so as to speed up future opening operations on the same block device file.

    d. Sets the inode->i_mapping field with the value of the corresponding field in the bdev inode. This is the pointer to the address space object, which will be explained in the section "The address_space Object" in Chapter 15.

    e. Inserts inode into the list of opened inodes of the block device descriptor rooted at bdev->bd_inodes.

    f. Returns the address bdev of the descriptor.

2. Sets the filp->i_mapping field with the value of inode->i_mapping (see step 1(d) above).

3. Gets the address of the gendisk descriptor relative to this block device:

        disk = get_gendisk(bdev->bd_dev, &part);

    If the block device being opened is a partition, the function also returns its index in the part local variable; otherwise, part is set to zero. The get_gendisk() function

simply invokes kobj_lookup() on the bdev_map kobject mapping domain passing the major and minor number of the device (see also the section "Device Driver Registration and Initialization" earlier in this chapter).

4. If bdev->bd_openers is not equal to zero, the block device has already been opened. Checks the bdev->bd_contains field:

    a. If it is equal to bdev, the block device is a whole disk: invokes the bdev->bd_disk->fops->open block device method, if defined, then checks the bdev->bd_invalidated field and invokes, if necessary, the rescan_partitions() functions (see comments on steps 6a and 6c later).

    b. If it not equal to bdev, the block device is a partition: increases the bdev->bd_contains->bd_part_count counter.

    Then, jumps to step 8.

5. Here the block device is being accessed for the first time. Initializes bdev->bd_disk with the address disk of the gendisk descriptor.

6. If the block device is a whole disk (part is zero), it executes the following substeps:

    a. If defined, it executes the disk->fops->open block device method: it is a custom function defined by the block device driver to perform any specific last-minute initialization.

    b. Gets from the hardsect_size field of the disk->queue request queue the sector size in bytes, and uses this value to set properly the bdev->bd_block_size and bdev->bd_inode->i_blkbits fields. Sets also the bdev->bd_inode->i_size field with the size of the disk computed from disk->capacity.

    c. If the bdev->bd_invalidated flag is set, it invokes rescan_partitions() to scan the partition table and update the partition descriptors. The flag is set by the check_disk_change block device method, which applies only to removable devices.

7. Otherwise if the block device is a partition (part is not zero), it executes the following substeps:

    a. Invokes bdget() again—this time passing the disk->first_minor minor number—to get the address whole of the block descriptor for the whole disk.

    b. Repeats steps from 3 to 6 for the block device descriptor of the whole disk, thus initializing it if necessary.

    c. Sets bdev->bd_contains to the address of the descriptor of the whole disk.

    d. Increases whole->bd_part_count to account for the new open operation on the partition of the disk.

    e. Sets bdev->bd_part with the value in disk->part[part-1]; it is the address of the hd_struct descriptor of the partition. Also, executes kobject_get(&bdev->bd_part->kobj) to increase the reference counter of the partition.

f. As in step 6b, sets the inode fields that specify size and sector size of the partition.

8. Increases the `bdev->bd_openers` counter.

9. If the block device file is being opened in exclusive mode (`O_EXCL` flag in `filp->f_flags` set), it invokes `bd_claim(bdev, filp)` to set the holder of the block device (see the section "Block Devices" earlier in this chapter). In case of error—block device has already an holder—it releases the block device descriptor and returns the error code `-EBUSY`.

10. Terminates by returning 0 (success).

Once the `blkdev_open()` function terminates, the `open()` system call proceeds as usual. Every future system call issued on the opened file will trigger one of the default block device file operations. As we will see in Chapter 16, each data transfer to or from the block device is effectively implemented by submitting requests to the generic block layer.

The Page Cache

As already mentioned in the section "The Common File Model" in Chapter 12, a disk cache is a software mechanism that allows the system to keep in RAM some data that is normally stored on a disk, so that further accesses to that data can be satisfied quickly without accessing the disk.

Disk caches are crucial for system performance, because repeated accesses to the same disk data are quite common. A User Mode process that interacts with a disk is entitled to ask repeatedly to read or write the same disk data. Moreover, different processes may also need to address the same disk data at different times. As an example, you may use the *cp* command to copy a text file and then invoke your favorite editor to modify it. To satisfy your requests, the command shell will create two different processes that access the same file at different times.

We have already encountered other disk caches in Chapter 12: the dentry cache, which stores dentry objects representing filesystem pathnames, and the inode cache, which stores inode objects representing disk inodes. Notice, however, that dentry objects and inode objects are not mere buffers storing the contents of some disk blocks; thus, the dentry cache and the inode cache are rather peculiar as disk caches.

This chapter deals with the page cache, which is a disk cache working on whole pages of data. We introduce the page cache in the first section. Then, we discuss in the section "Storing Blocks in the Page Cache" how the page cache can be used to retrieve single blocks of data (for instance, superblocks and inodes); this feature is crucial to speed up the VFS and the disk-based filesystems. Next, we describe in the section "Writing Dirty Pages to Disk" how the dirty pages in the page cache are written back to disk. Finally, we mention in the last section "The sync(), fsync(), and fdatasync() System Calls" some system calls that allow a user to flush the contents of the page cache so as to update the disk contents.

The Page Cache

The *page cache* is the main disk cache used by the Linux kernel. In most cases, the kernel refers to the page cache when reading from or writing to disk. New pages are added to the page cache to satisfy User Mode processes's read requests. If the page is not already in the cache, a new entry is added to the cache and filled with the data read from the disk. If there is enough free memory, the page is kept in the cache for an indefinite period of time and can then be reused by other processes without accessing the disk.

Similarly, before writing a page of data to a block device, the kernel verifies whether the corresponding page is already included in the cache; if not, a new entry is added to the cache and filled with the data to be written on disk. The I/O data transfer does not start immediately: the disk update is delayed for a few seconds, thus giving a chance to the processes to further modify the data to be written (in other words, the kernel implements deferred write operations).

Kernel code and kernel data structures don't need to be read from or written to disk.[*] Thus, the pages included in the page cache can be of the following types:

- Pages containing data of regular files; in Chapter 16, we describe how the kernel handles read, write, and memory mapping operations on them.
- Pages containing directories; as we'll see in Chapter 18, Linux handles the directories much like regular files.
- Pages containing data directly read from block device files (skipping the filesystem layer); as discussed in Chapter 16, the kernel handles them using the same set of functions as for pages containing data of regular files.
- Pages containing data of User Mode processes that have been swapped out on disk. As we'll see in Chapter 17, the kernel could be forced to keep in the page cache some pages whose contents have been already written on a swap area (either a regular file or a disk partition).
- Pages belonging to files of special filesystems, such as the *shm* special filesystem used for Interprocess Communication (IPC) shared memory region (see Chapter 19).

As you can see, each page included in the page cache contains data belonging to some file. This file—or more precisely the file's inode—is called the page's *owner*. (As we will see in Chapter 17, pages containing swapped-out data have the same owner even if they refer to different swap areas.)

---

[*] Well, almost never: if you want to resume the whole state of the system after a shutdown, you can perform a "suspend to disk" operation (*hibernation*), which saves the content of the whole RAM on a swap partition. We won't further discuss this case.

Practically all `read()` and `write()` file operations rely on the page cache. The only exception occurs when a process opens a file with the `O_DIRECT` flag set: in this case, the page cache is bypassed and the I/O data transfers make use of buffers in the User Mode address space of the process (see the section "Direct I/O Transfers" in Chapter 16); several database applications make use of the `O_DIRECT` flag so that they can use their own disk caching algorithm.

Kernel designers have implemented the page cache to fulfill two main requirements:

- Quickly locate a specific page containing data relative to a given owner. To take the maximum advantage from the page cache, searching it should be a very fast operation.

- Keep track of how every page in the cache should be handled when reading or writing its content. For instance, reading a page from a regular file, a block device file, or a swap area must be performed in different ways, thus the kernel must select the proper operation depending on the page's owner.

The unit of information kept in the page cache is, of course, a whole page of data. As we'll see in Chapter 18, a page does not necessarily contain physically adjacent disk blocks, so it cannot be identified by a device number and a block number. Instead, a page in the page cache is identified by an owner and by an index within the owner's data—usually, an inode and an offset inside the corresponding file.

The address_space Object

The core data structure of the page cache is the `address_space` object, a data structure embedded in the inode object that owns the page.[*] Many pages in the cache may refer to the same owner, thus they may be linked to the same `address_space` object. This object also establishes a link between the owner's pages and a set of methods that operate on these pages.

Each page descriptor includes two fields called `mapping` and `index`, which link the page to the page cache (see the section "Page Descriptors" in Chapter 8). The first field points to the `address_space` object of the inode that owns the page. The second field specifies the offset in page-size units within the owner's "address space," that is, the position of the page's data inside the owner's disk image. These two fields are used when looking for a page in the page cache.

Quite surprisingly, the page cache may happily contain multiple copies of the same disk data. For instance, the same 4-KB block of data of a regular file can be accessed in the following ways:

---

[*] An exception occurs for pages that have been swapped out. As we will see in Chapter 17, these pages have a common `address_space` object not included in any inode.

- Reading the file; hence, the data is included in a page owned by the regular file's inode.
- Reading the block from the device file (disk partition) that hosts the file; hence, the data is included in a page owned by the master inode of the block device file.

Thus, the same disk data appears in two different pages referenced by two different address_space objects.

The fields of the address_space object are shown in Table 15-1.

Table 15-1. The fields of the address_space object

Type	Field	Description
struct inode *	host	Pointer to the inode hosting this object, if any
struct radix_tree_root	page_tree	Root of radix tree identifying the owner's pages
spinlock_t	tree_lock	Spin lock protecting the radix tree
unsigned int	i_mmap_writable	Number of shared memory mappings in the address space
struct prio_tree_root	i_mmap	Root of the radix priority search tree (see Chapter 17)
struct list_head	i_mmap_nonlinear	List of non-linear memory regions in the address space
spinlock_t	i_mmap_lock	Spin lock protecting the radix priority search tree
unsigned int	truncate_count	Sequence counter used when truncating the file
unsigned long	nrpages	Total number of owner's pages
unsigned long	writeback_index	Page index of the last write-back operation on the owner's pages
struct address_space_operations *	a_ops	Methods that operate on the owner's pages
unsigned long	flags	Error bits and memory allocator flags
struct backing_dev_info *	backing_dev_info	Pointer to the backing_dev_info of the block device holding the data of this owner
spinlock_t	private_lock	Usually, spin lock used when managing the private_list list
struct list head	private_list	Usually, a list of dirty buffers of indirect blocks associated with the inode
struct address_space *	assoc_mapping	Usually, pointer to the address_space object of the block device including the indirect blocks

If the owner of a page in the page cache is a file, the address_space object is embedded in the i_data field of a VFS inode object. The i_mapping field of the inode always points to the address_space object of the owner of the pages containing the inode's data. The host field of the address_space object points to the inode object in which the descriptor is embedded.

Thus, if a page belongs to a file that is stored in an Ext3 filesystem, the owner of the page is the inode of the file and the corresponding `address_space` object is stored in the `i_data` field of the VFS inode object. The `i_mapping` field of the inode points to the `i_data` field of the same inode, and the `host` field of the `address_space` object points to the same inode.

Sometimes, however, things are more complicated. If a page contains data read from a block device file—that is, it stores "raw" data of a block device—the `address_space` object is embedded in the "master" inode of the file in the *bdev* special filesystem associated with the block device (this inode is referenced by the `bd_inode` field of the block device descriptor, see the section "Block Devices" in Chapter 14). Therefore, the `i_mapping` field of an inode of a block device file points to the `address_space` object embedded in the master inode; correspondingly, the `host` field of the `address_space` object points to the master inode. In this way, all pages containing data read from a block device have the same `address_space` object, even if they have been accessed by referring to different block device files.

The `i_mmap`, `i_mmap_writable`, `i_mmap_nonlinear`, and `i_mmap_lock` fields refer to memory mapping and reverse mapping. We'll discuss these topics in Chapter 16 and Chapter 17.

The `backing_dev_info` field points the `backing_dev_info` descriptor associated with the block device storing the data of the owner. As explained in the section "Request Queue Descriptors" in Chapter 14, the `backing_dev_info` structure is usually embedded in the request queue descriptor of the block device.

The `private_list` field is the head of a generic list that can be freely used by the filesystem for its specific purposes. For example, the Ext2 filesystem makes use of this list to collect the dirty buffers of "indirect" blocks associated with the inode (see the section "Data Blocks Addressing" in Chapter 18). When a flush operation forces the inode to be written to disk, the kernel flushes also all the buffers in this list. Moreover, the Ext2 filesystem stores in the `assoc_mapping` field a pointer to the `address_space` object of the block device containing the indirect blocks; it also uses the `assoc_mapping->private_lock` spin lock to protect the lists of indirect blocks in multiprocessor systems.

A crucial field of the `address_space` object is `a_ops`, which points to a table of type `address_space_operations` containing the methods that define how the owner's pages are handled. These methods are shown in Table 15-2.

Table 15-2. The methods of the address_space object

Method	Description
writepage	Write operation (from the page to the owner's disk image)
readpage	Read operation (from the owner's disk image to the page)
sync_page	Start the I/O data transfer of already scheduled operations on owner's pages
writepages	Write back to disk a given number of dirty owner's pages

Table 15-2. The methods of the address_space object (continued)

Method	Description
set_page_dirty	Set an owner's page as dirty
readpages	Read a list of owner's pages from disk
prepare_write	Prepare a write operation (used by disk-based filesystems)
commit_write	Complete a write operation (used by disk-based filesystems)
bmap	Get a logical block number from a file block index
invalidatepage	Invalidate owner's pages (used when truncating the file)
releasepage	Used by journaling filesystems to prepare the release of a page
direct_IO	Direct I/O transfer of the owner's pages (bypassing the page cache)

The most important methods are `readpage`, `writepage`, `prepare_write`, and `commit_write`. We discuss them in Chapter 16. In most cases, the methods link the owner inode objects with the low-level drivers that access the physical devices. For instance, the function that implements the `readpage` method for an inode of a regular file knows how to locate the positions on the physical disk device of the blocks corresponding to each page of the file. In this chapter, however, we don't have to discuss the `address_space` methods further.

The Radix Tree

In Linux, files can have large sizes, even a few terabytes. When accessing a large file, the page cache may become filled with so many of the file's pages that sequentially scanning all of them would be too time-consuming. In order to perform page cache lookup efficiently, Linux 2.6 makes use of a large set of search trees, one for each `address_space` object.

The `page_tree` field of an `address_space` object is the root of a *radix tree*, which contains pointers to the descriptors of the owner's pages. Given a page index denoting the position of the page inside the owner's disk image, the kernel can perform a very fast lookup operation in order to determine whether the required page is already included in the page cache. When looking up the page, the kernel interprets the index as a path inside the radix tree and quickly reaches the position where the page descriptor is—or should be—stored. If found, the kernel can retrieve from the radix tree the descriptor of the page; it can also quickly determine whether the page is dirty (i.e., to be flushed to disk) and whether an I/O transfer for its data is currently ongoing.

Each node of the radix tree can have up to 64 pointers to other nodes or to page descriptors. Nodes at the bottom level store pointers to page descriptors (the leaves), while nodes at higher levels store pointers to other nodes (the children). Each node is represented by the `radix_tree_node` data structure, which includes three fields: `slots` is an array of 64 pointers, `count` is a counter of how many pointers in the node are

not NULL, and `tags` is a two-component array of flags that will be discussed in the section "The Tags of the Radix Tree" later in this chapter. The root of the tree is represented by a `radix_tree_root` data structure, having three fields: `height` denotes the current tree's height (number of levels excluding the leaves), `gfp_mask` specifies the flags used when requesting memory for a new node, and `rnode` points to the `radix_tree_node` data structure corresponding to the node at level 1 of the tree (if any).

Let us consider a simple example. If none of the indices stored in the tree is greater than 63, the tree height is equal to one, because the 64 potential leaves can all be stored in the node at level 1 (see Figure 15-1 (a)). If, however, a new page descriptor corresponding to index 131 must be stored in the page cache, the tree height is increased to two, so that the radix tree can pinpoint indices up to 4095 (see Figure 15-1(b)).
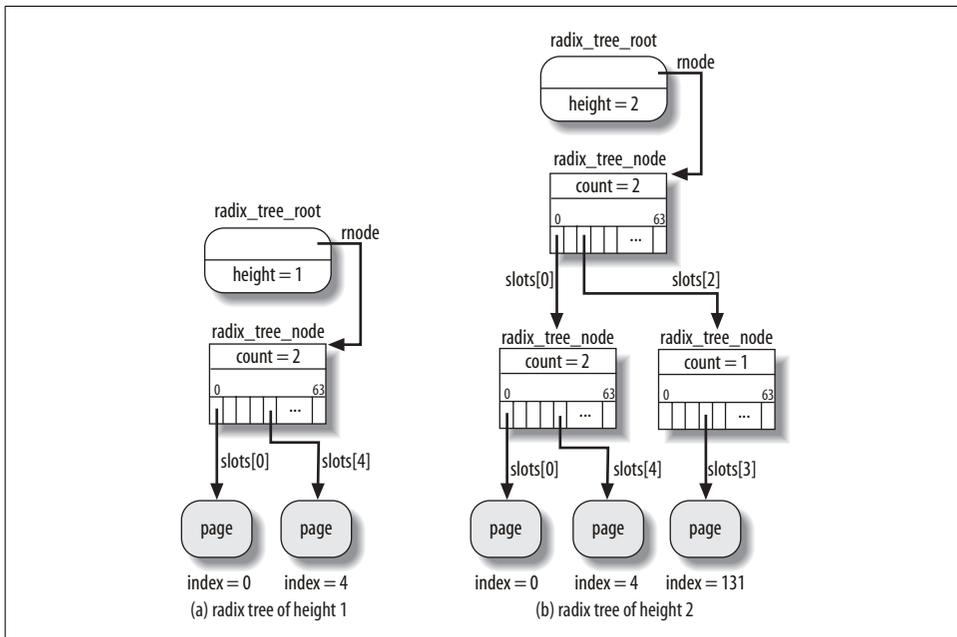
Figure 15-1. Two examples of a radix tree

Table 15-3 shows the highest page index and the corresponding maximum file size for each given height of the radix tree on a 32-bit architecture. In this case, the maximum height of a radix tree is six, although it is quite unlikely that the page cache of your system will make use of a radix tree that huge. Because the page index is stored in a 32-bit variable, when the tree has height equal to six, the node at the highest level can have at most four children.

Table 15-3. Highest index and maximum file size for each radix tree height

Radix tree height	Highest index	Maximum file size
0	none	0 bytes
1	$2^6 - 1 = 63$	256 kilobytes
2	$2^{12} - 1 = 4\,095$	16 megabytes
3	$2^{18} - 1 = 262\,143$	1 gigabyte
4	$2^{24} - 1 = 16\,777\,215$	64 gigabytes
5	$2^{30} - 1 = 1\,073\,741\,823$	4 terabytes
6	$2^{32} - 1 = 4\,294\,967\,295$	16 terabytes

The best way to understand how page lookup is performed is to recall how the paging system makes use of the page tables to translate linear addresses into physical addresses. As discussed in the section "Regular Paging" in Chapter 2, the 20 most significant bits of a linear address are split into two 10-bit long fields: the first field is an offset in the Page Directory, while the second one is an offset in the Page Table pointed to by the proper Page Directory entry.

A similar approach is used in the radix tree. The equivalent of the linear address is the page's index. However, the number of fields to be considered in the page's index depends on the height of the radix tree. If the radix tree has height 1, only indices ranging from 0 to 63 can be represented, thus the 6 less significant bits of the page's index are interpreted as the `slots` array index for the single node at level 1. If the radix tree has height 2, the indices that can be represented range from 0 to 4095; the 12 less significant bits of the page's index are thus split in 2 fields of 6 bits each; the most significant field is used as an array index for the node at level 1, while the less significant field is used as an array index for the node at level 2. The procedure is similar for every other radix tree's height. If the height is equal to 6, the 2 most significant bits of the page's index are the array index for the node at level 1, the following 6 bits are the array index for the node at level 2, and so on up to the 6 less significant bits, which are the array index for the node at level 6.

If the highest index of a radix tree is smaller than the index of a page that should be added, then the kernel increases the tree height correspondingly; the intermediate nodes of the radix tree depend on the value of the page index (see Figure 15-1 for an example).

Page Cache Handling Functions

The basic high-level functions that use the page cache involve finding, adding, and removing a page. Another function based on the previous ones ensures that the cache includes an up-to-date version of a given page.

Finding a page

The `find_get_page()` function receives as its parameters a pointer to an `address_space` object and an offset value. It acquires the address space's spin lock and invokes the `radix_tree_lookup()` function to search for a leaf node of the radix tree having the required offset. This function, in turn, starts from the root node of the tree and goes down according to the bits of the offset value, as explained in the previous section. If a `NULL` pointer is encountered, the function returns `NULL`; otherwise, it returns the address of a leaf node, that is, the pointer of the required page descriptor. If the requested page is found, `find_get_page()` increases its usage counter, releases the spin lock, and returns its address; otherwise, the function releases the spin lock and returns `NULL`.

The `find_get_pages()` function is similar to `find_get_page()`, but it performs a page cache lookup for a group of pages having contiguous indices. It receives as its parameters a pointer to an `address_space` object, the offset in the address space from where to start searching, the maximum number of pages to be retrieved, and a pointer to an array of pages descriptors to be filled by the function. To perform the lookup operation, `find_get_pages()` relies on the `radix_tree_gang_lookup()` function, which fills the array of pointers and returns the number of pages found. The returned pages have ascending indices, although there may be holes in the indices because some pages may not be in the page cache.

There are several other functions that perform search operations on the page cache. For example, the `find_lock_page()` function is similar to `find_get_page()`, but it increases the usage counter of the returned page and invokes `lock_page()` to set the `PG_locked` flag—thus, when the function returns, the page can be accessed exclusively by the caller. The `lock_page()` function, in turn, blocks the current process if the page is already locked. To that end, it invokes the `__wait_on_bit_lock()` function on the `PG_locked` bit. The latter function puts the current process in the `TASK_UNINTERRUPTIBLE` state, stores the process descriptor in a wait queue, executes the `sync_page` method of the `address_space` object to unplug the request queue of the block device containing the file, and finally invokes `schedule()` to suspend the process until the `PG_locked` flag of the page is cleared. To unlock a page and wake up the processes sleeping in the wait queue, the kernel makes use of the `unlock_page()` function.

The `find_trylock_page()` function is similar to `find_lock_page()`, except that it never blocks: if the requested page is already locked, the function returns an error code. Finally, the `find_or_create_page()` function executes `find_lock_page()`; however, if the page is not found, a new page is allocated and inserted in the page cache.

Adding a page

The `add_to_page_cache()` function inserts a new page descriptor in the page cache. It receives as its parameters the address `page` of the page descriptor, the address `mapping` of an `address_space` object, the value `offset` representing the page index inside the

address space, and the memory allocation flags `gfp_mask` to be used when allocating the new nodes of the radix tree. The function performs the following operations:

1. Invokes `radix_tree_preload()`, which disables kernel preemption and fills the per-CPU variable `radix_tree_preloads` with a few free `radix_tree_node` structures. Allocation of `radix_tree_node` structures is done by means of the `radix_tree_node_cachep` slab allocator cache. If `radix_tree_preload()` fails in preallocating the `radix_tree_node` structures, the `add_to_page_cache()` function terminates by returning the error code `-ENOMEM`. Otherwise, if `radix_tree_preload()` succeeds, `add_to_page_cache()` can be sure that the insertion of the new page descriptor will not fail for lack of free memory, at least for files of size up to 64 GB.

2. Acquires the `mapping->tree_lock` spin lock—notice that kernel preemption has already been disabled by `radix_tree_preload()`.

3. Invokes `radix_tree_insert()` to insert the new node in the tree. This function performs the following steps:

   a. Invokes `radix_tree_maxindex()` to get the maximum index that can be inserted in the radix tree with its current height; if the index of the new page cannot be represented with the current height, it invokes `radix_tree_extend()` to increase the height of the tree by adding the proper number of nodes (for instance, when applied to the radix tree shown in Figure 15-1 (a), `radix_tree_extend()` would add a single node on top of it). New nodes are allocated by executing the `radix_tree_node_alloc()` function, which tries to get a `radix_tree_node` structure from the slab allocator cache or, if this allocation fails, from the pool of preallocated structures stored in `radix_tree_preloads`.

   b. Starting from the root (`mapping->page_tree`), it traverses the tree according to the `offset` page's index until the leaf is reached, as described in the previous section. If required, it allocates new intermediate nodes by invoking `radix_tree_node_alloc()`.

   c. Stores the page descriptor address in the proper slot of the last traversed node of the radix tree, and returns 0.

4. Increases the usage counter `page->_count` of the page descriptor.

5. Because the page is new, its content is invalid: the function sets the `PG_locked` flag of the page frame to protect the page against concurrent accesses from other kernel control paths.

6. Initializes `page->mapping` and `page->index` with the parameters `mapping` and `offset`.

7. Increases the counter of cached pages in the address space (`mapping->nrpages`).

8. Releases the address space's spin lock.

9. Invokes `radix_tree_preload_end()` to reenable kernel preemption.

10. Returns 0 (success).

Removing a page

The `remove_from_page_cache()` function removes a page descriptor from the page cache. This is achieved in the following way:

1. Acquires the `page->mapping->tree_lock` spin lock and disables interrupts.

2. Invokes `radix_tree_delete()` to delete the node from the tree. This function receives as its parameters the address of the tree's root (`page->mapping->page_tree`) and the index of the page to be removed and performs the following steps:

    a. Starting from the root, it traverses the tree according to the page's index until the leaf is reached, as described in the previous section. While doing so, it builds up an array of `radix_tree_path` structures that describe the components of the path from the root to the leaf corresponding to the page to be deleted.

    b. Starts a cycle on the nodes collected in the path array, starting with the last node, which contains the pointer to the page descriptor. For each node, it sets to NULL the element of the slots array pointing to the next node (or to the page descriptor) and decreases the count field. If count becomes zero, it removes the node from the tree and releases the `radix_tree_node` structure to the slab allocator cache, then continues the cycle with the preceding node in the path array; otherwise, if count does not become zero, it continues with the next step.

    c. Returns the pointer to the page descriptor that has been removed from the tree.

3. Sets the `page->mapping` field to NULL.

4. Decreases by one the `page->mapping->nrpages` counter of cached pages.

5. Releases the `page->mapping->tree_lock` spin lock, enables the interrupts, and terminates.

Updating a page

The `read_cache_page()` function ensures that the cache includes an up-to-date version of a given page. Its parameters are a pointer `mapping` to an `address_space` object, an offset value `index` that specifies the requested page, a pointer `filler` to a function that reads the page's data from disk (usually it is the function that implements the address space's `readpage` method), and a pointer `data` that is passed to the `filler` function (usually, it is NULL). Here is a simplified description of what the function does:

1. Invokes `find_get_page()` to check whether the page is already in the page cache.

2. If the page is not in the page cache, it performs the following substeps:

    a. Invokes `alloc_pages()` to allocate a new page frame.

    b. Invokes `add_to_page_cache()` to insert the corresponding page descriptor into the page cache.

c. Invokes `lru_cache_add()` to insert the page in the zone's inactive LRU list (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).

3. Here the page is in the page cache. Invokes `mark_page_accessed()` to record the fact that the page has been accessed (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).

4. If the page is not up-to-date (`PG_uptodate` flag clear), it invokes the `filler` function to read from disk the page.

5. Returns the address of the page descriptor.

The Tags of the Radix Tree

As stated previously, the page cache not only allows the kernel to quickly retrieve a page containing specified data of a block device; the cache also allows the kernel to quickly retrieve pages in the cache that are in a given state.

For instance, let us suppose that the kernel must retrieve all pages in the cache that belong to a given owner and that are dirty, that is, the pages whose contents have not yet been written to disk. The `PG_dirty` flag stored in the page descriptor specifies whether a page is dirty or not; however, traversing the whole radix tree to sequentially access all the leaves—that is, the page descriptors—would be an unduly slow operation if most pages are not dirty.

Instead, to allow a quick search of dirty pages, each intermediate node in the radix tree contains a dirty tag for each child node (or leaf); this flag is set if and only if at least one of the dirty tags of the child node is set. The dirty tags of the nodes at the bottom level are usually copies of the `PG_dirty` flags of the page descriptors. In this way, when the kernel traverses a radix tree looking for dirty pages, it can skip each subtree rooted at an intermediate node whose dirty tag is clear: it knows for sure that all page descriptors stored in the subtree are not dirty.

The same idea applies to the `PG_writeback` flag, which denotes that a page is currently being written back to disk. Thus, each node of the radix tree propagates two flags of the page descriptor: `PG_dirty` and `PG_writeback` (see the section "Page Descriptors" in Chapter 8). To store them, each node includes two arrays of 64 bits in the `tags` field. The `tags[0]` array (`PAGECACHE_TAG_DIRTY`) is the dirty tag, while the `tags[1]` (`PAGECACHE_TAG_WRITEBACK`) array is the writeback tag.

The `radix_tree_tag_set()` function is invoked when setting the `PG_dirty` or the `PG_writeback` flag of a cached page; it acts on three parameters: the root of the radix tree, the page's index, and the type of tag to be set (`PAGECACHE_TAG_DIRTY` or `PAGECACHE_TAG_WRITEBACK`). The function starts from the root of the tree and goes down to the leaf corresponding to the given index; for each node of the path leading from the root to the leaf, the function sets the tag associated with the pointer to the next node in the path. The function then returns the address of the page descriptor. As a result, each in node in the path that goes down from the root to the leaf is tagged in the appropriate way.

The `radix_tree_tag_clear()` function is invoked when clearing the `PG_dirty` or the `PG_writeback` flag of a cached page; it acts on the same parameters as `radix_tree_tag_set()`. The function starts from the root of the tree and goes down to the leaf, building an array of `radix_tree_path` structures describing the path. Then, the function proceeds backward from the leaf to the root: it clears the tag of the node at the bottom level, then it checks whether all tags in the node's array are now cleared; if so, the function clears the proper tag in the parent node at the upper level, checks whether all tags in that node are cleared, and so on. The function then returns the address of the page descriptor.

When a page descriptor is removed from a radix tree, the proper tags in the nodes belonging to the path from the root to the leaf must be updated. The `radix_tree_delete()` function does this properly (even if we omitted mentioning this fact in the previous section). The `radix_tree_insert()` function, however, doesn't update the tags, because each page descriptor inserted in the radix tree is supposed to have the `PG_dirty` and `PG_writeback` flags cleared. If necessary, the kernel may later invoke the `radix_tree_tag_set()` function.

The `radix_tree_tagged()` function takes advantage of the arrays of flags included in all nodes of the tree to test whether a radix tree includes at least one page in a given state. The function performs this task quite simply by executing the following code (`root` is a pointer to the `radix_tree_root` structure of the radix tree, and `tag` is the flag to be tested):

```
for (idx = 0; idx < 2; idx++) {
    if (root->rnode->tags[tag][idx])
        return 1;
}
return 0;
```

Because the tags of all nodes of the radix tree can be assumed to be properly updated, `radix_tree_tagged()` needs only to check the tags of the node at level 1. An example of use of such function occurs when determining whether an inode contains dirty pages to be written to disk. Notice that in each iteration the function tests whether any of the 32 flags stored in an `unsigned long` is set.

The `find_get_pages_tag()` function is similar to `find_get_pages()` except that it returns only pages that are tagged with the `tag` parameter. As we'll see in the section "Writing Dirty Pages to Disk," this function is crucial to quickly identify all the dirty pages of an inode.

Storing Blocks in the Page Cache

We have seen in the section "Block Devices Handling" in Chapter 14 that the VFS, the mapping layer, and the various filesystems group the disk data in logical units called "blocks."

In old versions of the Linux kernel, there were two different main disk caches: the page cache, which stored whole pages of disk data resulting from accesses to the contents of the disk files, and the *buffer cache*, which was used to keep in memory the contents of the blocks accessed by the VFS to manage the disk-based filesystems.

Starting from stable version 2.4.10, the buffer cache does not really exist anymore. In fact, for reasons of efficiency, block buffers are no longer allocated individually; instead, they are stored in dedicated pages called "buffer pages," which are kept in the page cache.

Formally, a *buffer page* is a page of data associated with additional descriptors called "buffer heads," whose main purpose is to quickly locate the disk address of each individual block in the page. In fact, the chunks of data stored in a page belonging to the page cache are not necessarily adjacent on disk.

Block Buffers and Buffer Heads

Each block buffer has a *buffer head* descriptor of type `buffer_head`. This descriptor contains all the information needed by the kernel to know how to handle the block; thus, before operating on each block, the kernel checks its buffer head. The fields of a buffer head are listed in Table 15-4.

Table 15-4. The fields of a buffer head

Type	Field	Description
unsigned long	b_state	Buffer status flags
struct buffer_head *	b_this_page	Pointer to the next element in the buffer page's list
struct page *	b_page	Pointer to the descriptor of the buffer page holding this block
atomic_t	b_count	Block usage counter
u32	b_size	Block size
sector_t	b_blocknr	Block number relative to the block device (logical block number)
char *	b_data	Position of the block inside the buffer page
struct block_device *	b_bdev	Pointer to block device descriptor
bh_end_io_t *	b_end_io	I/O completion method
void *	b_private	Pointer to data for the I/O completion method
struct list_head	b_assoc_buffers	Pointers for the list of indirect blocks associated with an inode (see the section "The address_space Object" earlier in this chapter)

Two fields of the buffer head encode the disk address of the block: the `b_bdev` field identifies the block device—usually, a disk or a partition—that contains the block (see the section "Block Devices" in Chapter 14), while the `b_blocknr` field stores the *logical block number*, that is, the index of the block inside its disk or partition.

The b_data field specifies the position of the block buffer inside the buffer page. Actually, the encoding of this position depends on whether the page is in high memory or not. If the page is in high memory, the b_data field contains the offset of the block buffer with respect to the beginning of the page; otherwise, b_data contains the linear address of the block buffer.

The b_state field may store several flags. Some of them are of general use and are listed in Table 15-5. Each filesystem may also define its own private buffer head flags.

Table 15-5. The buffer head's general flags

Flag	Description
BH_Uptodate	Set if the buffer contains valid data
BH_Dirty	Set if the buffer is dirty—that is, it contains data that must be written to the block device
BH_Lock	Set if the buffer is locked, which usually happens when the buffer is involved in a disk transfer
BH_Req	Set if data transfer for initializing the buffer has already been requested
BH_Mapped	Set if the buffer is mapped to disk—that is, if the b_bdev and b_blocknr fields of the corresponding buffer head are significant
BH_New	Set if the corresponding block has just been allocated and has never been accessed
BH_Async_Read	Set if the buffer is being read asynchronously
BH_Async_Write	Set if the buffer is being written asynchronously
BH_Delay	Set if the buffer is not yet allocated on disk
BH_Boundary	Set if the block to be submitted after this one will not be adjacent to this one
BH_Write_EIO	Set if there was an I/O error when writing this block
BH_Ordered	Set if the block should be written strictly after the blocks submitted before it (used by journaling filesystems)
BH_Eopnotsupp	Set if the block device driver does not support the operation requested

Managing the Buffer Heads

The buffer heads have their own slab allocator cache, whose kmem_cache_s descriptor is stored in the bh_cachep variable. The alloc_buffer_head( ) and free_buffer_head( ) functions are used to get and release a buffer head, respectively.

The b_count field of the buffer head is a usage counter for the corresponding block buffer. The counter is increased right before each operation on the block buffer and decreased right after. The block buffers kept in the page cache are examined both periodically and when free memory becomes scarce, and only the block buffers having null usage counters may be reclaimed (see Chapter 17).

When a kernel control path wishes to access a block buffer, it should first increase the usage counter. The function that locates a block inside the page cache (__getblk( ); see the section "Searching Blocks in the Page Cache" later in this chapter) does this automatically, hence the higher-level functions do not usually increase the block buffer's usage counter.

When a kernel control path stops accessing a block buffer, it should invoke either __brelse() or __bforget() to decrease the corresponding usage counter. The difference between these two functions is that __bforget() also removes the block from any list of indirect blocks (b_assoc_buffers buffer head field; see the previous section "Block Buffers and Buffer Heads") and marks the buffer as clean, thus forcing the kernel to forget any change in the buffer that has yet to be written on disk.

Buffer Pages

Whenever the kernel must individually address a block, it refers to the buffer page that holds the block buffer and checks the corresponding buffer head.

Here are two common cases in which the kernel creates buffer pages:

- When reading or writing pages of a file that are not stored in contiguous disk blocks. This happens either because the filesystem has allocated noncontiguous blocks to the file, or because the file contains "holes" (see the section "File Holes" in Chapter 18).
- When accessing a single disk block (for instance, when reading a superblock or an inode block).

In the first case, the buffer page's descriptor is inserted in the radix tree of a regular file. The buffer heads are preserved because they store precious information: the block device and the logical block number that specify the position of the data in the disk. We will see how the kernel makes use of this type of buffer page in Chapter 16.

In the second case, the buffer page's descriptor is inserted in the radix tree rooted at the address_space object of the inode in the *bdev* special filesystem associated with the block device (see the section "The address_space Object" earlier in this chapter). This kind of buffer pages must satisfy a strong constraint: all the block buffers must refer to adjacent blocks of the underlying block device.

An instance of where this is useful is when the VFS wants to read the 1,024-byte inode block containing the inode of a given file. Instead of allocating a single buffer, the kernel must allocate a whole page storing four buffers; these buffers will contain the data of a group of four adjacent blocks on the block device, including the requested inode block.

In this chapter we will focus our attention on the second type of buffer pages, the so-called *block device buffer pages* (sometimes shortened to *blockdev's pages*).

All the block buffers within a single buffer page must have the same size; hence, on the 80 × 86 architecture, a buffer page can include from one to eight buffers, depending on the block size.

When a page acts as a buffer page, all buffer heads associated with its block buffers are collected in a singly linked circular list. The private field of the descriptor of the

buffer page points to the buffer head of the first block in the page;[*] every buffer head stores in the b_this_page field a pointer to the next buffer head in the list. Moreover, every buffer head stores the address of the buffer page's descriptor in the b_page field. Figure 15-2 shows a buffer page containing four block buffers and the corresponding buffer heads.
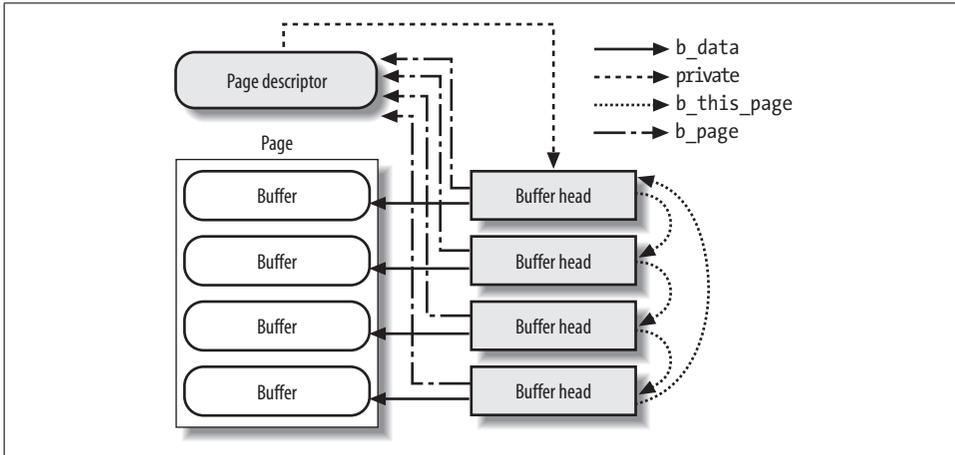
Figure 15-2. A buffer page including four buffers and their buffer heads

Allocating Block Device Buffer Pages

The kernel allocates a new block device buffer page when it discovers that the page cache does not include a page containing the buffer for a given block (see the section "Searching Blocks in the Page Cache" later in this chapter). In particular, the lookup operation for the block might fail for the following reasons:

1. The radix tree of the block device does not include a page containing the data of the block: in this case a new page descriptor must be added to the radix tree.

2. The radix tree of the block device includes a page containing the data of the block, but this page is not a buffer page: in this case new buffer heads must be allocated and linked to the page, thus transforming it into a block device buffer page.

3. The radix tree of the block device includes a buffer page containing the data of the block, but the page has been split in blocks of size different from the size of the requested block: in this case the old buffer heads must be released, and a new set of buffer heads must be allocated and linked to the page.

---

[*] Because the private field contains valid data, the PG_private flag of the page is also set; hence, if the page contains disk data and the PG_private flag is set, then the page is a buffer page. Notice, however, that other kernel components not related to the block I/O subsystem use the private and PG_private fields for other purposes.

In order to add a block device buffer page to the page cache, the kernel invokes the `grow_buffers()` function, which receives three parameters that identify the block:

- The address `bdev` of the `block_device` descriptor
- The logical block number `block`—the position of the block inside the block device
- The block size `size`

The function essentially performs the following actions:

1. Computes the offset `index` of the page of data within the block device that includes the requested block.

2. Invokes `grow_dev_page()` to create a new block device buffer page, if necessary. In turn, this function performs the following substeps:

   a. Invokes `find_or_create_page()`, passing to it the `address_space` object of the block device (`bdev->bd_inode->i_mapping`), the page offset `index`, and the `GFP_NOFS` flag. As described in the earlier section "Page Cache Handling Functions," `find_or_create_page()` looks for the page in the page cache and, if necessary, inserts a new page in the cache.

   b. Now the required page is in the page cache, and the function has the address of its descriptor. The function checks its `PG_private` flag; if it is `NULL`, the page is not yet a buffer page (it has no associated buffer heads): it jumps to step 2e.

   c. The page is already a buffer page. Gets from the `private` field of its descriptor the address `bh` of the first buffer head, and checks whether the block size `bh->size` is equal to the size of the requested block; if so, the page found in the page cache is a valid buffer page: it jumps to step 2g.

   d. The page has blocks of the wrong size: it invokes `try_to_free_buffers()` (see the next section) to release the previous buffer heads of the buffer page.

   e. Invokes the `alloc_page_buffers()` function to allocate the buffer heads for the blocks of the requested size within the page and insert them into the singly linked circular list implemented by the `b_this_page` fields. Moreover, the function initializes the `b_page` fields of the buffer heads with the address of the page descriptor, and the `b_data` fields with the offset or linear address of the block buffer inside the page.

   f. Stores the address of the first buffer head in the `private` field, sets the `PG_private` field, and increases the usage counter of the page (the block buffers inside the page counts as a page user).

   g. Invokes the `init_page_buffers()` function to initialize the `b_bdev`, `b_blocknr`, and `b_bstate` fields of the buffer heads linked to the page. All blocks are adjacent on disk, hence the logical block numbers are consecutive and can be easily derived from `block`.

   h. Returns the page descriptor address.

3. Unlocks the page (the page was locked by `find_or_create_page()`).

4. Decreases the page's usage counter (again, the counter was increased by `find_or_create_page()`).

5. Returns 1 (success).

Releasing Block Device Buffer Pages

As we will see in Chapter 17, block device buffer pages are released when the kernel tries to get additional free memory. Clearly a buffer page cannot be freed if it contains dirty or locked buffers. To release buffer pages, the kernel invokes the `try_to_release_page()` function, which receives the address page of a page descriptor and performs the following actions:[*]

1. If the `PG_writeback` flag of the page is set, it returns 0 (no release is possible because the page is being written back to disk).

2. If defined, it invokes the `releasepage` method of the block device's `address_space` object. (The method is usually not defined for block devices.)

3. Invokes the `try_to_free_buffers()` function, and returns its error code.

In turn, the `try_to_free_buffers()` function scans the buffer heads linked to the buffer page; it performs essentially the following actions:

1. Checks the flags of all the buffer heads of buffers included in the page. If some buffer head has the `BH_Dirty` or `BH_Locked` flag set, the function terminates by returning 0 (failure): it is not possible to release the buffers.

2. If a buffer head is inserted in a list of indirect buffers (see the section "Block Buffers and Buffer Heads" earlier in this chapter), the function removes it from the list.

3. Clears the `PG_private` flag of the page descriptor, sets the `private` field to `NULL`, and decreases the page's usage counter.

4. Clears the `PG_dirty` flag of the page.

5. Invokes repeatedly `free_buffer_head()` on the buffer heads of the page to free all of them.

6. Returns 1 (success).

Searching Blocks in the Page Cache

When the kernel needs to read or write a single block of a physical device (for instance, a superblock), it must check whether the required block buffer is already included in the page cache. Searching the page cache for a given block buffer—specified by the

---

[*] The `try_to_release_page()` function can also be invoked on buffer pages owned by regular files.

address bdev of a block device descriptor and by a logical block number nr—is a three stage process:

1. Get a pointer to the `address_space` object of the block device containing the block (`bdev->bd_inode->i_mapping`).

2. Get the block size of the device (`bdev->bd_block_size`), and compute the index of the page that contains the block. This is always a bit shift operation on the logical block number. For instance, if the block size is 1,024 bytes, each buffer page contains four block buffers, thus the page's index is `nr/4`.

3. Searches for the buffer page in the radix tree of the block device. After obtaining the page descriptor, the kernel has access to the buffer heads that describe the status of the block buffers inside the page.

Details are slightly more complicated than this, however. In order to enhance system performance, the kernel manages a `bh_lrus` array of small disk caches, one for each CPU, called the *Least Recently Used (LRU) block cache*. Each disk cache contains eight pointers to buffer heads that have been recently accessed by a given CPU. The elements in each CPU array are sorted so that the pointer to the most recently used buffer head has index 0. The same buffer head might appear on several CPU arrays (but never twice in the same CPU array); for each occurrence of a buffer head in the LRU block cache, the buffer head's `b_count` usage counter is increased by one.

The __find_get_block( ) function

The `__find_get_block()` function receives as its parameters the address bdev of a `block_device` descriptor, the block number block, and the block size size, and returns the address of the buffer head associated with the block buffer inside the page cache, or NULL if no such block buffer exists. The function performs essentially the following actions:

1. Checks whether the LRU block cache array of the executing CPU includes a buffer head whose `b_bdev`, `b_blocknr`, and `b_size` fields are equal to bdev, block, and size, respectively.

2. If the buffer head is in the LRU block cache, it reshuffles the elements in the array so as to put the pointer to the just discovered buffer head in the first position (index 0), increases its `b_count` field, and jumps to step 8.

3. Here the buffer head is not in the LRU block cache: it derives from the block number and the block size the page index relative to the block device as:

   ```
   index = block >> (PAGE_SHIFT - bdev->bd_inode->i_blkbits);
   ```

4. Invokes `find_get_page()` to locate, in the page cache, the descriptor of the buffer page containing the required block buffer. The function passes as parameters a pointer to the `address_space` object of the block device (`bdev->bd_inode->i_mapping`) and the page index to locate in the page cache the descriptor of the buffer page containing the required block buffer. If there is no such page in the cache, returns NULL (failure).

5. At this point, the function has the address of a descriptor for the buffer page: it scans the list of buffer heads linked to the buffer page, looking for the block having logical block number equal to `block`.

6. Decreases the `count` field of the page descriptor (it was increased by `find_get_page()`).

7. Moves all elements in the LRU block cache one position down, and inserts the pointer to the buffer head of the requested block in the first position. If a buffer head has been dropped out of the LRU block cache, it decreases its `b_count` usage counter.

8. Invokes `mark_page_accessed()` to move the buffer page in the proper LRU list, if necessary (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).

9. Returns the buffer head pointer.

The __getblk() function

The `__getblk()` function receives the same parameters as `__find_get_block()`, namely the address `bdev` of a `block_device` descriptor, the block number `block`, and the block size `size`, and returns the address of a buffer head associated with the buffer. The function never fails: even if the block does not exist at all, the `__getblk()` obligingly allocates a block device buffer page and returns a pointer to the buffer head that should describe the block. Notice that the block buffer returned by `__getblk()` does not necessarily contain valid data—the `BH_Uptodate` flag of the buffer head might be cleared.

The `__getblk()` function essentially performs the following steps:

1. Invokes `__find_get_block()` to check whether the block is already in the page cache. If the block is found, the function returns the address of its buffer head.

2. Otherwise, it invokes `grow_buffers()` to allocate a new buffer page for the requested block (see the section "Allocating Block Device Buffer Pages" earlier in this chapter).

3. If `grow_buffers()` fails in allocating such a page, `__getblk()` tries to reclaim some memory by invoking `free_more_memory()` (see Chapter 17).

4. Jumps back to step 1.

The __bread() function

The `__bread()` function receives the same parameters as `__getblk()`, namely the address `bdev` of a `block_device` descriptor, the block number `block`, and the block size `size`, and returns the address of a buffer head associated with the buffer. Contrary to `__getblk()`, the function reads the block from disk, if necessary, before returning the buffer head. The `__bread()` function performs the following steps:

1. Invokes `__getblk()` to find in the page cache the buffer page associated with the required block and to get a pointer to the corresponding buffer head.

2. If the block is already in the page cache and the buffer contains valid data (flag BH_Uptodate set), it returns the address of the buffer head.

3. Otherwise, it increases the usage counter of the buffer head.

4. Sets the b_end_io field to the address of end_buffer_read_sync() (see the next section).

5. Invokes submit_bh() to transmit the buffer head to the generic block layer (see next section).

6. Invokes wait_on_buffer() to put the current process in a wait queue until the read I/O operation is completed, that is, until the BH_Lock flag of the buffer head is cleared.

7. Returns the address of the buffer head.

Submitting Buffer Heads to the Generic Block Layer

A couple of functions, submit_bh() and ll_rw_block(), allow the kernel to start an I/O data transfer on one or more buffers described by their buffer heads.

The submit_bh( ) function

To transmit a single buffer head to the generic block layer, and thus to require the transfer of a single block of data, the kernel makes use of the submit_bh() function. Its parameters are the direction of data transfer (essentially READ or WRITE) and a pointer bh to the buffer head describing the block buffer.

The submit_bh() function assumes that the buffer head is fully initialized; in particular, the b_bdev, b_blocknr, and b_size fields must be properly set to identify the block on disk containing the requested data. If the block buffer belongs to a block device buffer page, the initialization of the buffer head is done by __find_get_block(), as described in the previous section. However, as we will see in the next chapter, submit_bh() can also be invoked on blocks belonging to buffer pages owned by regular files.

The submit_bh() function is little else than a glue function that creates a bio request from the contents of the buffer head and then invokes generic_make_request() (see the section "Submitting a Request" in Chapter 14). The main steps performed by it are the following:

1. Sets the BH_Req flag of the buffer head to record that the block has been submitted at least one time; moreover, if the direction of the data transfer is WRITE, clears the BH_Write_EIO flag.

2. Invokes bio_alloc() to allocate a new bio descriptor (see the section "The Bio Structure" in Chapter 14).

3. Initializes the fields of the bio descriptor according to the contents of the buffer head:

a. Sets the `bi_sector` field to the number of the first sector in the block (`bh->b_blocknr * bh->b_size / 512`);

b. Sets the `bi_bdev` field with the address of the block device descriptor (`bh->b_bdev`);

c. Sets the `bi_size` field with the block size (`bh->b_size`);

d. Initializes the first element of the `bi_io_vec` array so that the segment corresponds to the block buffer: `bi_io_vec[0].bv_page` is set to `bh->b_page`, `bi_io_vec[0].bv_len` is set to `bh->b_size`, and `bi_io_vec[0].bv_offset` is set to the offset of the block buffer in the page as specified by `bh->b_data`;

e. Sets `bi_vcnt` to 1 (just one segment on the bio), and `bi_idx` to 0 (the current segment to be transferred);

f. Sets the `bi_end_io` field to the address of `end_bio_bh_io_sync()`, and sets the `bi_private` field to the address of the buffer head; the function will be invoked when the data transfer terminates (see below).

4. Increases the reference counter of the bio (it becomes equal to 2).

5. Invokes `submit_bio()`, which sets the `bi_rw` flag with the direction of the data transfer, updates the `page_states` per-CPU variable to keep track of the number of sectors read and written, and invokes the `generic_make_request()` function on the bio descriptor.

6. Decreases the usage counter of the bio; the bio descriptor is not freed, because it is now inserted in a queue of the I/O scheduler.

7. Returns 0 (success).

When the I/O data transfer on the bio terminates, the kernel executes the `bi_end_io` method, in this particular case the `end_bio_bh_io_sync()` function. The latter function essentially gets the address of the buffer head from the `bi_private` field of the bio, then invokes the `b_end_io` method of the buffer head—it was properly set before invoking `submit_bh()`—and finally invokes `bio_put()` to destroy the bio structure.

The ll_rw_block() function

Sometimes the kernel must trigger the data transfer of several data blocks at once, which are not necessarily physically adjacent. The `ll_rw_block()` function receives as its parameters the direction of data transfer (essentially READ or WRITE), the number of blocks to be transferred, and an array of pointers to buffer heads describing the corresponding block buffers. The function iterates over all buffer heads; for each of them, it executes the following actions:

1. Tests and sets the `BH_Lock` flag of the buffer head; if the buffer was already locked, the data transfer has been activated by another kernel control path, so just skips the buffer by jumping to step 9.

2. Increases by one the usage counter `b_count` of the buffer head.

3. If the data transfer direction is `WRITE`, it sets the `b_end_io` method of the buffer head to point to the address of the `end_buffer_write_sync()` function; otherwise, it sets the `b_end_io` method to point to the address of the `end_buffer_read_sync()` function.

4. If the data transfer direction is `WRITE`, it tests and clears the `BH_Dirty` flag of the buffer head. If the flag was not set, there is no need to write the block on disk, so it jumps to step 7.

5. If the data transfer direction is `READ` or `READA` (read-ahead), it checks whether the `BH_Uptodate` flag of the buffer head is set; if so, there is no need to read the block from disk, so it jumps to step 7.

6. Here the block has to be read or written: it invokes the `submit_bh()` function to pass the buffer head to the generic block layer, then jumps to step 9.

7. Unlocks the buffer head by clearing the `BH_Lock` flag, and awakens every process that was waiting for the block being unlocked.

8. Decreases the `b_count` field of the buffer head.

9. If there are other buffer heads in the array to be processed, it selects the next one and jumps back to step 1; otherwise, it terminates.

Notice that if the `ll_rw_block()` function passes a buffer head to the generic block layer, it leaves the buffer locked and its reference counter increased, so that the buffer cannot be accessed and cannot be freed until the data transfer completes. The kernel executes the `b_end_io` completion method of the buffer head when the data transfer for the block terminates. Assuming that there was no I/O error, the `end_buffer_write_sync()` and `end_buffer_read_sync()` functions simply set the `BH_Uptodate` field of the buffer head, unlock the buffer, and decrease its usage counter.

Writing Dirty Pages to Disk

As we have seen, the kernel keeps filling the page cache with pages containing data of block devices. Whenever a process modifies some data, the corresponding page is marked as dirty—that is, its `PG_dirty` flag is set.

Unix systems allow the deferred writes of dirty pages into block devices, because this noticeably improves system performance. Several write operations on a page in cache could be satisfied by just one slow physical update of the corresponding disk sectors. Moreover, write operations are less critical than read operations, because a process is usually not suspended due to delayed writings, while it is most often suspended because of delayed reads. Thanks to deferred writes, each physical block device will service, on the average, many more read requests than write ones.

A dirty page might stay in main memory until the last possible moment—that is, until system shutdown. However, pushing the delayed-write strategy to its limits has two major drawbacks:

- If a hardware or power supply failure occurs, the contents of RAM can no longer be retrieved, so many file updates that were made since the system was booted are lost.

- The size of the page cache, and hence of the RAM required to contain it, would have to be huge—at least as big as the size of the accessed block devices.

Therefore, dirty pages are *flushed* (written) to disk under the following conditions:

- The page cache gets too full and more pages are needed, or the number of dirty pages becomes too large.

- Too much time has elapsed since a page has stayed dirty.

- A process requests all pending changes of a block device or of a particular file to be flushed; it does this by invoking a `sync( )`, `fsync( )`, or `fdatasync( )` system call (see the section "The sync(), fsync(), and fdatasync() System Calls" later in this chapter).

Buffer pages introduce a further complication. The buffer heads associated with each buffer page allow the kernel to keep track of the status of each individual block buffer. The `PG_dirty` flag of the buffer page should be set if at least one of the associated buffer heads has the `BH_Dirty` flag set. When the kernel selects a dirty buffer page for flushing, it scans the associated buffer heads and effectively writes to disk only the contents of the dirty blocks. As soon as the kernel flushes all dirty blocks in a buffer page to disk, it clears the `PG_dirty` flag of the page.

The pdflush Kernel Threads

Earlier versions of Linux used a kernel thread called *bdflush* to systematically scan the page cache looking for dirty pages to flush, and they used a second kernel thread called *kupdate* to ensure that no page remains dirty for too long. Linux 2.6 has replaced both of them with a group of general purpose kernel threads called *pdflush*.

These kernel threads have a flexible structure. They act on two parameters: a pointer to a function to be executed by the thread and a parameter for the function. The number of *pdflush* kernel threads in the system is dynamically adjusted: new threads are created when they are too few and existing threads are killed when they are too many. Because the functions executed by these kernel threads can block, creating several *pdflush* kernel threads instead of a single one, leads to better system performance.

Births and deaths are governed by the following rules:

- There must be at least two *pdflush* kernel threads and at most eight.

- If there were no idle *pdflush* during the last second, a new *pdflush* should be created.

- If more than one second elapsed since the last *pdflush* became idle, a *pdflush* should be removed.

Each *pdflush* kernel thread has a `pdflush_work` descriptor (see Table 15-6). The descriptors of idle *pdflush* kernel threads are collected in the `pdflush_list` list; the `pdflush_lock` spin lock protects that list from concurrent accesses in multiprocessor systems. The `nr_pdflush_threads` variable[*] stores the total number of *pdflush* kernel threads (idle and busy). Finally, the `last_empty_jifs` variable stores the last time (in jiffies) since the `pdflush_list` list of *pdflush* threads became empty.

Table 15-6. The fields of the pdflush_work descriptor

Type	Field	Description
struct task_struct *	who	Pointer to kernel thread descriptor
void(*)(unsigned long)	fn	Callback function to be executed by the kernel thread
unsigned long	arg0	Argument to callback function
struct list head	list	Links for the `pdflush_list` list
unsigned long	when_i_went_ to_sleep	Time in jiffies when kernel thread became available

Each *pdflush* kernel thread executes the `__pdflush()` function, which essentially loops in an endless cycle until the kernel thread dies. Let's suppose that the *pdflush* kernel thread is idle; then, the process is sleeping in `TASK_INTERRUPTIBLE` state. As soon as the kernel thread is woken up, `__pdflush()` accesses its `pdflush_work` descriptor and executes the callback function stored in the `fn` field, passing to it the argument stored in the `arg0` field. When the callback function terminates, `__pdflush()` checks the value of the `last_empty_jifs` variable: if there was no idle *pdflush* kernel thread for more than one second and if there are less than eight *pdflush* kernel threads, `__pdflush()` starts another kernel thread. Otherwise, if the last entry in the `pdflush_list` list is idle for more than one second, and there are more than two *pdflush* kernel threads, `__pdflush()` terminates: as explained in the section "Kernel Threads" in Chapter 3, the corresponding kernel thread executes the `_exit()` system call and it is thus destroyed. Otherwise, `__pdflush()` reinserts the `pdflush_work` descriptor of the kernel thread in the `pdflush_list` list and puts the kernel thread to sleep.

The `pdflush_operation()` function is used to activate an idle *pdflush* kernel thread. This function acts on two parameters: a pointer `fn` to the function that must be executed and an argument `arg0`; it performs the following steps:

1. Extracts from the `pdflush_list` list a pointer `pdf` to the `pdflush_work` descriptor of an idle *pdflush* kernel thread. If the list is empty, it returns -1. If the list contained just one element, it sets the value of the `last_empty_jifs` variable to `jiffies`.

2. Stores in `pdf->fn` and in `pdf->arg0` the parameters `fn` and `arg0`.

---

[*] The value of this variable can be read from the */proc/sys/vm/nr_pdflush_threads* file.

3. Invokes `wake_up_process()` to wake up the idle *pdflush* kernel thread, that is, `pdf->who`.

What kinds of jobs are delegated to the *pdflush* kernel threads? There are a few of them, all related to flushing of dirty data. In particular, *pdflush* usually executes one of the following callback functions:

- `background_writeout()`: systematically walks the page cache looking for dirty pages to be flushed (see the next section "Looking for Dirty Pages To Be Flushed").
- `wb_kupdate()`: checks that no page in the page cache remains dirty for too long (see the section "Retrieving Old Dirty Pages" later in this chapter).

Looking for Dirty Pages To Be Flushed

Every radix tree could include dirty pages to be flushed. Retrieving all of them thus involves an exhaustive search among all `address_space` objects associated with inodes having an image on disk. Because the page cache might include a large number of pages, scanning the whole cache in a single run might keep the CPU and the disks busy for a long time. Therefore, Linux adopts a sophisticated mechanism that splits the page cache scanning in several runs of execution.

The `wakeup_bdflush()` function receives as argument the number of dirty pages in the page cache that should be flushed; the value zero means that all dirty pages in the cache should be written back to disk. The function invokes `pdflush_operation()` to wake up a *pdflush* kernel thread (see the previous section) and delegate to it the execution of the `background_writeout()` callback function. The latter function effectively retrieves the specified number of dirty pages from the page cache and writes them back to disk.

The `wakeup_bdflush()` function is executed when either memory is scarce or a user makes an explicit request for a flush operation. In particular, the function is invoked when:

- The User Mode process issues a `sync()` system call (see the section "The sync(), fsync(), and fdatasync() System Calls" later in this chapter).
- The `grow_buffers()` function fails to allocate a new buffer page (see the earlier section "Allocating Block Device Buffer Pages").
- The page frame reclaiming algorithm invokes `free_more_memory()` or `try_to_free_pages()` (see Chapter 17).
- The `mempool_alloc()` function fails to allocate a new memory pool element (see the section "Memory Pools" in Chapter 8).

Moreover, a *pdflush* kernel thread executing the `background_writeout()` callback function is woken up by every process that modifies the contents of pages in the page cache and causes the fraction of dirty pages to rise above some *dirty background*

threshold. The background threshold is typically set to 10% of all pages in the system, but its value can be adjusted by writing in the */proc/sys/vm/dirty_background_ratio* file.

The `background_writeout()` function relies on a `writeback_control` structure, which acts as a two-way communication device: on one hand, it tells an auxiliary function called `writeback_inodes()` what to do; on the other hand, it stores some statistics about the number of pages written to disk. The most important fields of this structure are the following:

`sync_mode`
> Specifies the synchronization mode: `WB_SYNC_ALL` means that if a locked inode is encountered, it must be waited upon and not just skipped over; `WB_SYNC_HOLD` means that locked inodes are put in a list for later consideration; and `WB_SYNC_NONE` means that locked inodes are simply skipped.

`bdi`
> If not `NULL`, it points to a `backing_dev_info` structure; in this case, only dirty pages belonging to the underlying block device will be flushed.

`older_than_this`
> If not null, it means that inodes younger than the specified value should be skipped.

`nr_to_write`
> Number of dirty pages yet to be written in this run of execution.

`nonblocking`
> If this flag is set, the process cannot be blocked.

The `background_writeout()` function acts on a single parameter: `nr_pages`, the minimum number of pages that should be flushed to disk. It essentially executes the following steps:

1. Reads from the `page_state` per-CPU variable the number of pages and dirty pages currently stored in the page cache. If the fraction of dirty pages is below a given threshold and at least `nr_pages` have been flushed to disk, the function terminates. The value of this threshold is typically set to about 40% of the number of pages in the system; it could be adjusted by writing into the */proc/sys/vm/dirty_ratio* file.

2. Invokes `writeback_inodes()` to try to write 1,024 dirty pages (see below).

3. Checks the number of pages effectively written and decreases the number of pages yet to be written.

4. If less than 1,024 pages have been written or if pages have been skipped, probably the request queue of the block device is congested: the function puts the current process to sleep in a special wait queue for 100 milliseconds or until the queue becomes uncongested.

5. Goes back to step 1.

The `writeback_inodes()` function acts on a single parameter, namely a pointer `wbc` to a `writeback_control` descriptor. The `nr_to_write` field of this descriptor contains the number of pages to be flushed to disk. When the function returns, the same field contains the number of pages remaining to be flushed; if everything went smoothly, this field will be set to 0.

Let us suppose that `writeback_inodes()` is called with the `wbc->bdi` and `wbc->older_than_this` pointers set to `NULL`, the `WB_SYNC_NONE` synchronization mode, and the `wbc->nonblocking` flag set—these are the values set by `background_writeout()`. The function scans the list of superblocks rooted at the `super_blocks` variable (see the section "Superblock Objects" in Chapter 12). The scanning ends when either the whole list has been traversed, or the target number of pages to be flushed has been reached. For each superblock `sb`, the function executes the following steps:

1. Checks whether the `sb->s_dirty` or `sb->s_io` lists are empty: the first list collects the dirty inodes of the superblock, while the second list collects the inodes waiting to be transferred to disk (see below). If both lists are empty, the inodes on this filesystem have no dirty pages, so the function considers the next superblock in the list.

2. Here the superblock has dirty inodes. Invokes `sync_sb_inodes()` on the `sb` superblock. This function:

   a. Puts all the inodes of `sb->s_dirty` into the list pointed to by `sb->s_io` and clears the list of dirty inodes.

   b. Gets the next `inode` pointer from `sb->s_io`. If this list is empty, it returns.

   c. If the inode was dirtied after `sync_sb_inodes()` started, it skips the inode's dirty pages and returns. Notice that some dirty inodes might remain in the `sb->s_io` list.

   d. If the current process is a *pdflush* kernel thread, it checks whether another *pdflush* kernel thread running on another CPU is already trying to flush dirty pages for files belonging to this block device. This can be done by an atomic test and set operation on the `BDI_pdflush` flag of the inode's `backing_dev_info`. Essentially, it is pointless to have more than one *pdflush* kernel thread on the same request queue (see the section "The pdflush Kernel Threads" earlier in this chapter).

   e. Increases by one the inode's usage counter.

   f. Invokes `__writeback_single_inode()` to write back the dirty buffers associated with the selected inode:

      1. If the inode is locked, it moves `inode` into the list of dirty inodes (`inode->i_sb->s_dirty`) and returns 0. (Since we are assuming that the `wbc->sync_mode` field is not `WB_SYNC_ALL`, the function does not block waiting for the inode to unlock.)

2. Uses the `writepages` method of the inode's address space, or the `mpage_writepages()` function if no such method exists, to write up to `wbc->nr_to_write` dirty pages. This function uses the `find_get_pages_tag()` function to retrieve quickly all dirty pages in the inode's address space (see the section "The Tags of the Radix Tree" earlier in this chapter). Details will be given in the next chapter.

3. If the inode is dirty, it uses the superblock's `write_inode` method to write the inode to disk. The functions that implement this method usually rely on `submit_bh()` to transfer a single block of data (see the section "Submitting Buffer Heads to the Generic Block Layer" earlier in this chapter).

4. Checks the status of the inode; accordingly, moves the inode back into the `sb->s_dirty` list if some page of the inode is still dirty, or in the `inode_unused` list if the inode's reference counter is zero, or in the `inode_in_use` list otherwise (see the section "Inode Objects" in Chapter 12).

5. Returns the error code of the function invoked in step 2f2.

g. Back into the `sync_sb_inodes()` function. If the current process is the *pdflush* kernel thread, it clears the `BDI_pdflush` flag set in step 2d.

h. If some pages were skipped in the inode just processed, then the inode includes locked buffers: moves all inodes remaining in the `sb->s_io` list back into the `sb->s_dirty` list: they will be reconsidered at a later time.

i. Decreases by one the usage counter of the inode.

j. If `wbc->nr_to_write` is greater than 0, goes back to step 2b to look for other dirty inodes of the same superblock. Otherwise, the `sync_sb_inodes()` function terminates.

3. Back into the `writeback_inodes()` function. If `wbc->nr_to_write` is greater than zero, it jumps to step 1 and continues with the next superblock in the global list. Otherwise, it returns.

Retrieving Old Dirty Pages

As stated earlier, the kernel tries to avoid the risk of starvation that occurs when some pages are not flushed for a long period of time. Hence, if a page remains dirty for a predefined amount of time, the kernel explicitly starts an I/O data transfer that writes its contents to disk.

The job of retrieving old dirty pages is delegated to a *pdflush* kernel thread that is periodically woken up. During the kernel initialization, the `page_writeback_init()` function sets up the `wb_timer` dynamic timer so that it decays after `dirty_writeback_centisecs` hundreds of a second (usually 500, but this value can be adjusted by writing in the */proc/sys/vm/dirty_writeback_centisecs* file). The timer function, which is

called `wb_timer_fn()`, essentially invokes the `pdflush_operation()` function passing to it the address of the `wb_kupdate()` callback function.

The `wb_kupdate()` function walks the page cache looking for "old" dirty inodes; it executes the following steps:

1. Invokes the `sync_supers()` function to write the dirty superblocks to disk (see the next section). Although not strictly related to the flushing of the pages in the page cache, this invocation ensures that no superblock remains dirty for more than, usually, five seconds.

2. Stores in the `older_than_this` field of a `writeback_control` descriptor a pointer to a value in jiffies corresponding to the current time minus 30 seconds. Thirty seconds is the longest time for which a page is allowed to remain dirty.

3. Determines from the per-CPU `page_state` variable the rough number of dirty pages currently in the page cache.

4. Invokes repeatedly `writeback_inodes()` until either the number of pages written to disk reaches the value determined in the previous step, or all pages older than 30 seconds have been written. During this cycle the function might sleep if some request queue becomes congested.

5. Uses `mod_timer()` to restart the `wb_timer` dynamic timer: it will decay once again `dirty_writeback_centisecs` hundreds of seconds since the invocation of this function (or one second since now if this execution lasted too long).

The sync( ), fsync( ), and fdatasync( ) System Calls

In this section, we examine briefly the three system calls available to user applications to flush dirty buffers to disk:

`sync( )`
    Allows a process to flush all dirty buffers to disk

`fsync( )`
    Allows a process to flush all blocks that belong to a specific open file to disk

`fdatasync( )`
    Very similar to `fsync()`, but doesn't flush the inode block of the file

The sync( ) System Call

The service routine `sys_sync()` of the `sync()` system call invokes a series of auxiliary functions:

```
wakeup_bdflush(0);
sync_inodes(0);
sync_supers();
sync_filesystems(0);
sync_filesystems(1);
sync_inodes(1);
```

As described in the previous section, `wakeup_bdflush()` starts a *pdflush* kernel thread, which flushes to disk all dirty pages contained in the page cache.

The `sync_inodes()` function scans the list of superblocks looking for dirty inodes to be flushed; it acts on a `wait` parameter that specifies whether it must wait until flushing has been performed or not. The function scans the superblocks of all currently mounted filesystems; for each superblock containing dirty inodes, `sync_inodes()` first invokes `sync_sb_inodes()` to flush the corresponding dirty pages (we described this function earlier in the section "Looking for Dirty Pages To Be Flushed"), then invokes `sync_blockdev()` to explicitly flush the dirty buffer pages owned by the block device that includes the superblock. This is done because the `write_inode` superblock method of many disk-based filesystems simply marks the block buffer corresponding to the disk inode as dirty; the `sync_blockdev()` function makes sure that the updates made by `sync_sb_inodes()` are effectively written to disk.

The `sync_supers()` function writes the dirty superblocks to disk, if necessary, by using the proper `write_super` superblock operations. Finally, the `sync_filesystems()` executes the `sync_fs` superblock method for all writable filesystems. This method is simply a hook offered to a filesystem in case it needs to perform some peculiar operation at each sync; this method is only used by journaling filesystems such as Ext3 (see Chapter 18).

Notice that `sync_inodes()` and `sync_filesystems()` are invoked twice, once with the `wait` parameter equal to 0 and the second time with the parameter equal to 1. This is done on purpose: first, they quickly flush to disk the unlocked inodes; next, they wait for each locked inode to become unlocked and finish writing them one by one.

The fsync() and fdatasync() System Calls

The `fsync()` system call forces the kernel to write to disk all dirty buffers that belong to the file specified by the `fd` file descriptor parameter (including the buffer containing its inode, if necessary). The corresponding service routine derives the address of the file object and then invokes the `fsync` method. Usually, this method ends up invoking the `__writeback_single_inode()` function to write back both the dirty pages associated with the selected inode and the inode itself (see the section "Looking for Dirty Pages To Be Flushed" earlier in this chapter).

The `fdatasync()` system call is very similar to `fsync()`, but writes to disk only the buffers that contain the file's data, not those that contain inode information. Because Linux 2.6 does not have a specific file method for `fdatasync()`, this system call uses the `fsync` method and is thus identical to `fsync()`.

Accessing Files

Accessing a disk-based file is a complex activity that involves the VFS abstraction layer (Chapter 12), handling block devices (Chapter 14), and the use of the page cache (Chapter 15). This chapter shows how the kernel builds on all those facilities to carry out file reads and writes. The topics covered in this chapter apply both to regular files stored in disk-based filesystems and to block device files; these two kinds of files will be referred to simply as "files."

The stage we are working at in this chapter starts after the proper read or write method of a particular file has been called (as described in Chapter 12). We show here how each read ends with the desired data delivered to a User Mode process and how each write ends with data marked ready for transfer to disk. The rest of the transfer is handled by the facilities described in Chapter 14 and Chapter 15.

There are many different ways to access a file. In this chapter we will consider the following cases:

Canonical mode
> The file is opened with the O_SYNC and O_DIRECT flags cleared, and its content is accessed by means of the read() and write() system calls. In this case, the read() system call blocks the calling process until the data is copied into the User Mode address space (however, the kernel is always allowed to return fewer bytes than requested!). The write() system call is different, because it terminates as soon as the data is copied into the page cache (deferred write). This case is covered in the section "Reading and Writing a File."

Synchronous mode
> The file is opened with the O_SYNC flag set—or the flag is set at a later time by the fcntl() system call. This flag affects only the write operation (read operations are always blocking), which blocks the calling process until the data is effectively written to disk. The section "Reading and Writing a File" covers this case, too.

Memory mapping mode
> After opening the file, the application issues an mmap() system call to map the file into memory. As a result, the file appears as an array of bytes in RAM, and the

application accesses directly the array elements instead of using read(), write(), or lseek(). This case is discussed in the section "Memory Mapping."

Direct I/O mode

The file is opened with the O_DIRECT flag set. Any read or write operation transfers data directly from the User Mode address space to disk, or vice versa, bypassing the page cache. We discuss this case in the section "Direct I/O Transfers." (The values of the O_SYNC and O_DIRECT flags can be combined in four meaningful ways.)

Asynchronous mode

The file is accessed—either through a group of POSIX APIs or by means of Linux-specific system calls—in such a way to perform "asynchronous I/O:" this means the requests for data transfers never block the calling process; rather, they are carried on "in the background" while the application continues its normal execution. We discuss this case in the section "Asynchronous I/O."

Reading and Writing a File

The section "The read() and write() System Calls" in Chapter 12 described how the read() and write() system calls are implemented. The corresponding service routines end up invoking the file object's read and write methods, which may be filesystem-dependent. For disk-based filesystems, these methods locate the physical blocks that contain the data being accessed and activate the block device driver to start the data transfer.

Reading a file is page-based: the kernel always transfers whole pages of data at once. If a process issues a read() system call to get a few bytes, and that data is not already in RAM, the kernel allocates a new page frame, fills the page with the suitable portion of the file, adds the page to the page cache, and finally copies the requested bytes into the process address space. For most filesystems, reading a page of data from a file is just a matter of finding what blocks on disk contain the requested data. Once this is done, the kernel fills the pages by submitting the proper I/O operations to the generic block layer. In practice, the read method of all disk-based filesystems is implemented by a common function named generic_file_read().

Write operations on disk-based files are slightly more complicated to handle, because the file size could increase, and therefore the kernel might allocate some physical blocks on the disk. Of course, how this is precisely done depends on the filesystem type. However, many disk-based filesystems implement their write methods by means of a common function named generic_file_write(). Examples of such filesystems are Ext2, System V/Coherent/Xenix, and MINIX. On the other hand, several other filesystems, such as journaling and network filesystems, implement the write method by means of custom functions.

Reading from a File

The `generic_file_read( )` function is used to implement the `read` method for block device files and for regular files of almost all disk-based filesystems. This function acts on the following parameters:

`filp`

   Address of the file object

`buf`

   Linear address of the User Mode memory area where the characters read from the file must be stored

`count`

   Number of characters to be read

`ppos`

   Pointer to a variable that stores the offset from which reading must start (usually the `f_pos` field of the `filp` file object)

As a first step, the function initializes two descriptors. The first descriptor is stored in the local variable `local_iov` of type `iovec`; it contains the address (`buf`) and the length (`count`) of the User Mode buffer that shall receive the data read from the file. The second descriptor is stored in the local variable `kiocb` of type `kiocb`; it is used to keep track of the completion status of an ongoing synchronous or asynchronous I/O operation. The main fields of the `kiocb` descriptor are shown in Table 16-1.

Table 16-1. The main fields of the kiocb descriptor

Type	Field	Description
`struct list_head`	`ki_run_list`	Pointers for the list of I/O operations to be retried later
`long`	`ki_flags`	Flags of the `kiocb` descriptor
`int`	`ki_users`	Usage counter of the `kiocb` descriptor
`unsigned int`	`ki_key`	Identifier of the asynchronous I/O operation, or `KIOCB_SYNC_KEY` (`0xffffffff`) for synchronous I/O operations
`struct file *`	`ki_filp`	Pointer to the file object associated with the ongoing I/O operation
`struct kioctx *`	`ki_ctx`	Pointer to the asynchronous I/O context descriptor for this operation (see the section "Asynchronous I/O" later in this chapter)
`int (*) (struct kiocb *, struct io_event *)`	`ki_cancel`	Method invoked when canceling an asynchronous I/O operation
`ssize_t (*) (struct kiocb *)`	`ki_retry`	Method invoked when retrying an asynchronous I/O operation
`void (*) (struct kiocb *)`	`ki_dtor`	Method invoked when destroying the `kiocb` descriptor
`struct list_head`	`ki_list`	Pointers for the list of active ongoing I/O operation on an asynchronous I/O context

Table 16-1. The main fields of the kiocb descriptor (continued)

Type	Field	Description
union	ki_obj	For synchronous operations, pointer to the process descriptor that issued the I/O operation; for asynchronous operations, pointer to the `iocb` User Mode data structure
__u64	ki_user_data	Value to be returned to the User Mode process
loff_t	ki_pos	Current file position of the ongoing I/O operation
unsigned short	ki_opcode	Type of operation (read, write, or sync)
size_t	ki_nbytes	Number of bytes to be transferred
char *	ki_buf	Current position in the User Mode buffer
size_t	ki_left	Number of bytes yet to be transferred
wait_queue_t	ki_wait	Wait queue used for asynchronous I/O operations
void *	private	Freely usable by the filesystem layer

The `generic_file_read()` function initializes the `kiocb` descriptor by executing the `init_sync_kiocb` macro, which sets the fields of the object for a synchronous operation. In particular, the macro sets the `ki_key` field to `KIOCB_SYNC_KEY`, the `ki_filp` field to `filp`, and the `ki_obj` field to `current`.

Then, `generic_file_read()` invokes `__generic_file_aio_read()` passing to it the addresses of the `iovec` and `kiocb` descriptors just filled. The latter function returns a value, which is usually the number of bytes effectively read from the file; `generic_file_read()` terminates by returning this value.

The `__generic_file_aio_read()` function is a general-purpose routine used by all filesystems to implement both synchronous and asynchronous read operations. The function receives four parameters: the address `iocb` of a `kiocb` descriptor, the address `iov` of an array of `iovec` descriptors, the length of this array, and the address `ppos` of a variable that stores the file's current pointer. When invoked by `generic_file_read()`, the array of `iovec` descriptors is composed of just one element describing the User Mode buffer that will receive the data.[*]

We now explain the actions of the `__generic_file_aio_read()` function; for the sake of simplicity, we restrict the description to the most common case: a synchronous operation raised by a `read()` system call on a page-cached file. Later in this chapter we describe how this function behaves in other cases. As usual, we do not discuss how errors and anomalous conditions are handled.

---

[*] A variant of the `read()` system call—named `readv()`—allows an application to define multiple User Mode buffers in which the kernel scatters the data read from the file; the `__generic_file_aio_read()` function handles this case, too. In the following, we will assume that the data read from the file will be copied into just one User Mode buffer; however, guessing the additional steps to be performed when using multiple buffers is straightforward.

Here are the steps performed by the function:

1. Invokes `access_ok()` to verify that the User Mode buffer described by the `iovec` descriptor is valid. Because the starting address and length have been received from the `sys_read()` service routine, they must be checked before using them (see the section "Verifying the Parameters" in Chapter 10). If the parameters are not valid, returns the `-EFAULT` error code.

2. Sets up a *read operation descriptor*—namely, a data structure of type `read_descriptor_t` that stores the current status of the ongoing file read operation relative to a single User Mode buffer. The fields of this descriptor are shown in Table 16-2.

3. Invokes `do_generic_file_read()`, passing to it the file object pointer `filp`, the pointer to the file offset `ppos`, the address of the just allocated read operation descriptor, and the address of the `file_read_actor()` function (see later).

4. Returns the number of bytes copied into the User Mode buffer; that is, the value found in the `written` field of the `read_descriptor_t` data structure.

Table 16-2. The fields of the read operation descriptor

Type	Field	Description
`size_t`	`written`	How many bytes have been copied into the User Mode buffer
`size_t`	`count`	How many bytes are yet to be transferred
`char *`	`arg.buf`	Current position in the User Mode buffer
`int`	`error`	Error code of the read operation (0 for no error)

The `do_generic_file_read()` function reads the requested pages from disk and copies them into the User Mode buffer. In particular, the function performs the following actions:

1. Gets the `address_space` object corresponding to the file being read; its address is stored in `filp->f_mapping`.

2. Gets the owner of the `address_space` object, that is, the inode object that will own the pages to be filled with file's data; its address is stored in the `host` field of the `address_space` object. If the file being read is a block device file, the owner is an inode in the *bdev* special filesystem rather than the inode pointed to by `filp->f_dentry->d_inode` (see "The address_space Object" in Chapter 15).

3. Considers the file as subdivided in pages of data (4,096 bytes per page). The function derives from the file pointer `*ppos` the logical number of the page that includes the first requested byte—that is, the page's index in the address space—and stores it in the `index` local variable. The function also stores in the `offset` local variable the displacement inside the page of the first requested byte.

4. Starts a cycle to read all pages that include the requested bytes; the number of bytes to be read is stored in the `count` field of the `read_descriptor_t` descriptor.

During a single iteration, the function transfers a page of data by performing the following substeps:

a. If index*4096+offset exceeds the file size stored in the i_size field of the inode object, it exits from the cycle and goes to step 5.

b. Invokes cond_resched() to check the TIF_NEED_RESCHED flag of the current process and, if the flag is set, to invoke the schedule() function.

c. If additional pages must be read in advance, it invokes page_cache_readahead() to read them. We defer discussing read-ahead until the later section "Read-Ahead of Files."

d. Invokes find_get_page() passing as parameters a pointer to the address_space object and the value of index; the function looks up the page cache to find the descriptor of the page that stores the requested data, if any.

e. If find_get_page() returned a NULL pointer, the page requested is not in the page cache. In that case, it performs the following actions:

   1. Invokes handle_ra_miss() to tune the parameters used by the read-ahead system.

   2. Allocates a new page.

   3. Inserts the descriptor of the new page into the page cache by invoking add_to_page_cache(). Remember that this function sets the PG_locked flag of the new page.

   4. Inserts the descriptor of the new page into the LRU list by invoking lru_cache_add() (see Chapter 17).

   5. Jumps to step 4j to start reading the file's data.

f. If the function has reached this point, the page is in the page cache. Checks the PG_uptodate flag; if it is set, the data stored in the page is up-to-date, hence there is no need to read it from disk: jumps to step 4m.

g. The data on the page is not valid, so it must be read from disk. The function gains exclusive access to the page by invoking the lock_page() function. As described in the section "Page Cache Handling Functions" in Chapter 15, lock_page() suspends the current process if the PG_locked flag is already set, until that bit is cleared.

h. Now the page is locked by the current process. However, another process might have removed the page from the page cache right before the previous step; hence, it checks whether the mapping field of the page descriptor is NULL; in this case, it unlocks the page by invoking unlock_page(), decreases its usage counter (it was increased by find_get_page()), and jumps back to step 4a starting over with the same page.

i. If the function has reached this point, the page is locked and still present in the page cache. Checks the PG_uptodate flag again, because another kernel control path could have completed the necessary read between steps 4f

and 4g. If the flag is set, it invokes `unlock_page()` and jumps to step 4m to skip the read operation.

j. Now the actual I/O operation can be started. Invokes the readpage method of the `address_space` object of the file. The corresponding function takes care of activating the I/O data transfer from the disk to the page. We discuss later what this function does for regular files and block device files.

k. If the `PG_uptodate` flag is still cleared, it waits until the page has been effectively read by invoking the `lock_page()` function. The page, which was locked in step 4g, will be unlocked as soon as the read operation finishes. Therefore, the current process sleeps until the I/O data transfer terminates.

l. If index exceeds the file size in pages (this number is obtained by dividing the value of the `i_size` field of the inode object by 4,096), it decreases the page's usage counter, and exits from the cycle jumping to step 5. This case occurs when the file being read is concurrently truncated by another process.

m. Stores in the `nr` local variable the number of bytes in the page that should be copied into the User Mode buffer. This value is equal to the page size (4,096 bytes) unless either `offset` is not zero—this can happen only for the first or last page of requested data—or the file does not contain all requested bytes.

n. Invokes `mark_page_accessed()` to set the `PG_referenced` or the `PG_active` flag, hence denoting the fact that the page is being used and should not be swapped out (see Chapter 17). If the same page (or part thereof) is read several times in successive executions of `do_generic_file_read()`, this step is executed only during the first read.

o. Now it is time to copy the data on the page into the User Mode buffer. To do this, `do_generic_file_read()` invokes the `file_read_actor()` function, whose address has been passed as a parameter. In turn, `file_read_actor()` essentially executes the following steps:

   1. Invokes `kmap()`, which establishes a permanent kernel mapping for the page if it is in high memory (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8).

   2. Invokes `__copy_to_user()`, which copies the data on the page in the User Mode address space (see the section "Accessing the Process Address Space" in Chapter 10). Notice that this operation might block the process because of page faults while accessing the User Mode address space.

   3. Invokes `kunmap()` to release any permanent kernel mapping of the page.

   4. Updates the `count`, `written`, and `buf` fields of the `read_descriptor_t` descriptor.

p. Updates the `index` and `offset` local variables according to the number of bytes effectively transferred in the User Mode buffer. Typically, if the last

byte in the page has been copied into the User Mode buffer, `index` is increased by one and `offset` is set to zero; otherwise, `index` is not increased and `offset` is set to the number of bytes in the page that have been copied into the User Mode buffer.

    q. Decreases the page descriptor usage counter.

    r. If the `count` field of the `read_descriptor_t` descriptor is not zero, there is other data to be read from the file: jumps to step 4a to continue the loop with the next page of data in the file.

5. All requested—or available—bytes have been read. The function updates the `filp->f_ra` read-ahead data structure to record the fact that data is being read sequentially from the file (see the later section "Read-Ahead of Files").

6. Assigns to `*ppos` the value `index*4096+offset`, thus storing the next position where a sequential access is to occur for a future invocation of the `read()` and `write()` system calls.

7. Invokes `update_atime()` to store the current time in the `i_atime` field of the file's inode and to mark the inode as dirty, and returns.

The readpage method for regular files

As we saw, the `readpage` method is used repeatedly by `do_generic_file_read()` to read individual pages from disk into memory.

The `readpage` method of the `address_space` object stores the address of the function that effectively activates the I/O data transfer from the physical disk to the page cache. For regular files, this field typically points to a wrapper that invokes the `mpage_readpage()` function. For instance, the `readpage` method of the Ext3 filesystem is implemented by the following function:

```
int ext3_readpage(struct file *file, struct page *page)
{
    return mpage_readpage(page, ext3_get_block);
}
```

The wrapper is needed because the `mpage_readpage()` function receives as its parameters the descriptor `page` of the page to be filled and the address `get_block` of a function that helps `mpage_readpage()` find the right block. The wrapper is filesystem-specific and can therefore supply the proper function to get a block. This function translates the block numbers relative to the beginning of the file into logical block numbers relative to positions of the block in the disk partition (for an example, see Chapter 18). Of course, the latter parameter depends on the type of filesystem to which the regular file belongs; in the previous example, the parameter is the address of the `ext3_get_block()` function. The function passed as `get_block` always uses a buffer head to store precious information about the block device (`b_dev` field), the position of the requested data on the device (`b_blocknr` field), and the block status (`b_state` field).

The `mpage_readpage()` function chooses between two different strategies when reading a page from disk. If the blocks that contain the requested data are contiguously located on disk, then the function submits the read I/O operation to the generic block layer by using a single bio descriptor. In the opposite case, each block in the page is read by using a different bio descriptor. The filesystem-dependent `get_block` function plays the crucial role of determining whether the next block in the file is also the next block on the disk.

Specifically, `mpage_readpage()` performs the following steps:

1. Checks the `PG_private` field of the page descriptor: if it is set, the page is a buffer page, that is, the page is associated with a list of buffer heads describing the blocks that compose the page (see the section "Storing Blocks in the Page Cache" in Chapter 15). This means that the page has already been read from disk in the past, and that the blocks in the page are not adjacent on disk: jumps to step 11 to read the page one block at a time.

2. Retrieves the block size (stored in the `page->mapping->host->i_blkbits` inode field), and computes two values required to access all blocks on that page: the number of blocks stored in the page and the file block number of the first block in the page—that is, the index of the first block in the page relative to the beginning of the file.

3. For each block in the page, invokes the filesystem-dependent `get_block` function passed as a parameter to get the logical block number, that is, the index of the block relative to the beginning of the disk or partition. The logical block numbers of all blocks in the page are stored in a local array.

4. Checks for any anomalous condition that could occur while executing the previous step. In particular, if some blocks are not adjacent on disk, or some block falls inside a "file hole" (see the section "File Holes" in Chapter 18), or a block buffer has been already filled by the `get_block` function, then jumps to step 11 to read the page one block at a time.

5. If the function has reached this point, all blocks on the page are adjacent on disk. However, the page could be the last page of data in the file, hence some of the blocks in the page might not have an image on disk. If so, it fills the corresponding block buffers in the page with zeros; otherwise, it sets the `PG_mappedtodisk` flag of the page descriptor.

6. Invokes `bio_alloc()` to allocate a new bio descriptor consisting of a single segment and to initialize its `bi_bdev` and `bi_sector` fields with the address of the block device descriptor and the logical block number of the first block in the page, respectively. Both pieces of information have been determined in step 3 above.

7. Sets the `bio_vec` descriptor of the bio's segment with the initial address of the page, the offset of the first byte to be read (zero), and the total number of bytes to be read.

8. Stores the address of the `mpage_end_io_read()` function in the `bio->bi_end_io` field (see below).

9. Invokes `submit_bio()`, which sets the `bi_rw` flag with the direction of the data transfer, updates the `page_states` per-CPU variable to keep track of the number of read sectors, and invokes the `generic_make_request()` function on the bio descriptor (see the section "Issuing a Request to the I/O Scheduler" in Chapter 14).

10. Returns the value zero (success).

11. If the function jumps here, the page contains blocks that are not adjacent on disk. If the page is up-to-date (`PG_uptodate` flag set), the function invokes `unlock_page()` to unlock the page; otherwise, it invokes `block_read_full_page()` to start reading the page one block at a time (see below).

12. Returns the value zero (success).

The `mpage_end_io_read()` function is the completion method of the bio; it is executed as soon as the I/O data transfer terminates. Assuming that there was no I/O error, the function essentially sets the `PG_uptodate` flag of the page descriptor, invokes `unlock_page()` to unlock the page and to wake up any process sleeping for this event, and invokes `bio_put()` to destroy the bio descriptor.

The readpage method for block device files

In the sections "VFS Handling of Device Files" in Chapter 13 and "Opening a Block Device File" in Chapter 14, we discussed how the kernel handles requests to open a block device file. We saw how the `init_special_inode()` function sets up the device inode and how the `blkdev_open()` function completes the opening phase.

Block devices use an `address_space` object that is stored in the `i_data` field of the corresponding block device inode in the *bdev* special filesystem. Unlike regular files—whose `readpage` method in the `address_space` object depends on the filesystem type to which the file belongs—the `readpage` method of block device files is always the same. It is implemented by the `blkdev_readpage()` function, which calls `block_read_full_page()`:

```
int blkdev_readpage(struct file * file, struct * page page)
{
    return block_read_full_page(page, blkdev_get_block);
}
```

As you can see, the function is once again a wrapper, this time for the `block_read_full_page()` function. This time the second parameter points to a function that translates the file block number relative to the beginning of the file into a logical block number relative to the beginning of the block device. For block device files, however, the two numbers coincide; therefore, the `blkdev_get_block()` function performs the following steps:

1. Checks whether the number of the first block in the page exceeds the index of the last block in the block device (this index is obtained by dividing the size of the block device stored in `bdev->bd_inode->i_size` by the block size stored in `bdev->bd_block_size`; `bdev` points to the descriptor of the block device). If so, it returns `-EIO` for a write operation, or zero for a read operation. (Reading beyond the end of a block device is not allowed, either, but the error code should not be returned here: the kernel could just be trying to dispatch a read request for the last data of a block device, and the corresponding buffer page is only partially mapped.)

2. Sets the `b_dev` field of the buffer head to `bdev`.

3. Sets the `b_blocknr` field of the buffer head to the file block number, which was passed as a parameter of the function.

4. Sets the `BH_Mapped` flag of the buffer head to state that the `b_dev` and `b_blocknr` fields of the buffer head are significant.

The `block_read_full_page()` function reads a page of data one block at a time. As we have seen, it is used both when reading block device files and when reading pages of regular files whose blocks are not adjacent on disk. It performs the following steps:

1. Checks the `PG_private` flag of the page descriptor; if it is set, the page is associated with a list of buffer heads describing the blocks that compose the page (see the section "Storing Blocks in the Page Cache" in Chapter 15). Otherwise, the function invokes `create_empty_buffers()` to allocate buffer heads for all block buffers included in the page. The address of the buffer head for the first buffer in the page is stored in the `page->private` field. The `b_this_page` field of each buffer head points to the buffer head of the next buffer in the page.

2. Derives from the file offset relative to the page (`page->index` field) the file block number of the first block in the page.

3. For each buffer head of the buffers in the page, it performs the following substeps:

   a. If the `BH_Uptodate` flag is set, it skips the buffer and continues with the next buffer in the page.

   b. If the `BH_Mapped` flag is not set and the block is not beyond the end of the file, it invokes the filesystem-dependent `get_block` function whose address has been passed as a parameter. For a regular file, the function looks in the on-disk data structures of the filesystem and finds the logical block number of the buffer relative to the beginning of the disk or partition. Conversely, for a block device file, the function regards the file block number as the logical block number. In both cases the function stores the logical block number in the `b_blocknr` field of the corresponding buffer head and sets the `BH_Mapped` flag.[*]

---

[*] When accessing a regular file, the `get_block` function might not find the block if it falls in a "file hole" (see the section "File Holes" in Chapter 18). In this case, the function fills the block buffer with zeros and sets the `BH_Uptodate` flag of the buffer head.

c. Tests again the `BH_Uptodate` flag because the filesystem-dependent `get_block` function could have triggered a block I/O operation that updated the buffer. If `BH_Uptodate` is set, it continues with the next buffer in the page.

d. Stores the address of the buffer head in `arr` local array, and continues with the next buffer in the page.

4. If no file hole has been encountered in the previous step, the function sets the `PG_mappedtodisk` flag of the page.

5. Now the `arr` local array stores the addresses of the buffer heads that correspond to the buffers whose content is not up-to-date. If this array is empty, all buffers in the page are valid. So the function sets the `PG_uptodate` flag of the page descriptor, unlocks the page by invoking `unlock_page()`, and returns.

6. The `arr` local array is not empty. For each buffer head in the array, `block_read_full_page()` performs the following substeps:

a. Sets the `BH_Lock` flag. If the flag was already set, the function waits until the buffer is released.

b. Sets the `b_end_io` field of the buffer head to the address of the `end_buffer_async_read()` function (see below) and sets the `BH_Async_Read` flag of the buffer head.

7. For each buffer head in the `arr` local array, it invokes the `submit_bh()` function on it, specifying the operation type READ. As we saw earlier, this function triggers the I/O data transfer of the corresponding block.

8. Returns 0.

The `end_buffer_async_read()` function is the completion method of the buffer head; it is executed as soon as the I/O data transfer on the block buffer terminates. Assuming that there was no I/O error, the function sets the `BH_Uptodate` flag of the buffer head and clears the `BH_Async_Read` flag. Then, the function gets the descriptor of the buffer page containing the block buffer (its address is stored in the `b_page` field of the buffer head) and checks whether all blocks in the page are up-to-date; if so, the function sets the `PG_uptodate` flag of the page and invokes `unlock_page()`.

Read-Ahead of Files

Many disk accesses are sequential. As we will see in Chapter 18, regular files are stored on disk in large groups of adjacent sectors, so that they can be retrieved quickly with few moves of the disk heads. When a program reads or copies a file, it often accesses it sequentially, from the first byte to the last one. Therefore, many adjacent sectors on disk are likely to be fetched when handling a series of a process's read requests on the same file.

Read-ahead consists of reading several adjacent pages of data of a regular file or block device file *before* they are actually requested. In most cases, read-ahead significantly enhances disk performance, because it lets the disk controller handle fewer

commands, each of which refers to a larger chunk of adjacent sectors. Moreover, it improves system responsiveness. A process that is sequentially reading a file does not usually have to wait for the requested data because it is already available in RAM.

However, read-ahead is of no use when an application performs random accesses to files; in this case, it is actually detrimental because it tends to waste space in the page cache with useless information. Therefore, the kernel reduces—or stops—read-ahead when it determines that the most recently issued I/O access is not sequential to the previous one.

Read-ahead of files requires a sophisticated algorithm for several reasons:

- Because data is read page by page, the read-ahead algorithm does not have to consider the offsets inside the page, but only the positions of the accessed pages inside the file.

- Read-ahead may be gradually increased as long as the process keeps accessing the file sequentially.

- Read-ahead must be scaled down or even disabled when the current access is not sequential with respect to the previous one (random access).

- Read-ahead should be stopped when a process keeps accessing the same pages over and over again (only a small portion of the file is being used), or when almost all pages of the file are already in the page cache.

- The low-level I/O device driver should be activated at the proper time, so that the future pages will have been transferred when the process needs them.

The kernel considers a file access as *sequential* with respect to the previous file access if the first page requested is the page following the last page requested in the previous access.

While accessing a given file, the read-ahead algorithm makes use of two sets of pages, each of which corresponds to a contiguous portion of the file. These two sets are called the *current window* and the *ahead window*.

The current window consists of pages requested by the process or read in advance by the kernel and included in the page cache. (A page in the current window is not necessarily up-to-date, because its I/O data transfer could still be in progress.) The current window contains both the last pages sequentially accessed by the process and possibly some of the pages that have been read in advance by the kernel but that have not yet been requested by the process.

The ahead window consists of pages—following the ones in the current window—that are being currently being read in advance by the kernel. No page in the ahead window has yet been requested by the process, but the kernel assumes that sooner or later the process will request them.

When the kernel recognizes a sequential access and the initial page belongs to the current window, it checks whether the ahead window has already been set up. If not,

the kernel creates a new ahead window and triggers the read operations for the corresponding pages. In the ideal case, the process still requests pages from the current window while the pages in the ahead window are being transferred. When the process requests a page included in the ahead window, the ahead window becomes the new current window.

The main data structure used by the read-ahead algorithm is the `file_ra_state` descriptor whose fields are listed in Table 16-3. Each file object includes such a descriptor in its `f_ra` field.

Table 16-3. The fields of the file_ra_state descriptor

Type	Field	Description
unsigned long	start	Index of first page in the current window
unsigned long	size	Number of pages included in the current window (-1 for read-ahead temporarily disabled, 0 for empty current window)
unsigned long	flags	Flags used to control the read-ahead
unsigned long	cache_hit	Number of consecutive cache hits (pages requested by the process and found in the page cache)
unsigned long	prev_page	Index of the last page requested by the process
unsigned long	ahead_start	Index of the first page in the ahead window
unsigned long	ahead_size	Number of pages in the ahead window (0 for an empty ahead window)
unsigned long	ra_pages	Maximum size in pages of a read-ahead window (0 for read-ahead permanently disabled)
unsigned long	mmap_hit	Read-ahead hit counter (used for memory mapped files)
unsigned long	mmap_miss	Read-ahead miss counter (used for memory mapped files)

When a file is opened, all the fields of its `file_ra_state` descriptor are set to zero except the `prev_page` and `ra_pages` fields.

The `prev_page` field stores the index of the last page requested by the process in the previous read operation; initially, the field contains the value -1.

The `ra_pages` field represents the maximum size in pages for the current window, that is, the maximum read-ahead allowed for the file. The initial (default) value for this field is stored in the `backing_dev_info` descriptor of the block device that includes the file (see the section "Request Queue Descriptors" in Chapter 14). An application can tune the read-ahead algorithm for a given opened file by modifying the `ra_pages` field; this can be done by invoking the `posix_fadvise()` system call, passing to it the commands `POSIX_FADV_NORMAL` (set read-ahead maximum size to default, usually 32 pages), `POSIX_FADV_SEQUENTIAL` (set read-ahead maximum size to two times the default), and `POSIX_FADV_RANDOM` (set read-ahead maximum size to zero, thus permanently disabling read-ahead).

The flags field contains two flags called RA_FLAG_MISS and RA_FLAG_INCACHE that play an important role. The first flag is set when a page that has been read in advance is not found in the page cache (likely because it has been reclaimed by the kernel in order to free memory; see Chapter 17): in this case, the size of the next ahead window to be created is somewhat reduced. The second flag is set when the kernel determines that the last 256 pages requested by the process have all been found in the page cache (the value of consecutive cache hits is stored in the ra->cache_hit field). In this case, read-ahead is turned off because the kernel assumes that all the pages required by the process are already in the cache.

When is the read-ahead algorithm executed? This happens in the following cases:

- When the kernel handles a User Mode request to read pages of file data; this event triggers the invocation of the page_cache_readahead() function (see step 4c in the description of the do_generic_file_read() function in the section "Reading from a File" earlier in this chapter).

- When the kernel allocates a page for a file memory mapping (see the filemap_nopage() function in the section "Demand Paging for Memory Mapping" later in this chapter, which again invokes the page_cache_readahead() function).

- When a User Mode application executes the readahead() system call, which explicitly triggers some read-ahead activity on a file descriptor.

- When a User Mode application executes the posix_fadvise() system call with the POSIX_FADV_NOREUSE or POSIX_FADV_WILLNEED commands, which inform the kernel that a given range of file pages will be accessed in the near future.

- When a User Mode application executes the madvise() system call with the MADV_WILLNEED command, which informs the kernel that a given range of pages in a file memory mapping region will be accessed in the near future.

The page_cache_readahead() function

The page_cache_readahead() function takes care of all read-ahead operations that are not explicitly triggered by ad-hoc system calls. It replenishes the current and ahead windows, updating their sizes according to the number of read-ahead hits, that is, according to how successful the read-ahead strategy was in the past accesses to the file.

The function is invoked when the kernel must satisfy a read request for one or more pages of a file, and acts on five parameters:

mapping
    Pointer to the address_space object that describes the owner of the page

ra
    Pointer to the file_ra_state descriptor of the file containing the page

filp
    Address of the file object

`offset`

    Offset of the page within the file

`req_size`

    Number of pages yet to be read to complete the current read operation[*]

Figure 16-1 shows the flow diagram of `page_cache_readahead()`. The function essentially acts on the fields of the `file_ra_state` descriptor; thus, although the description of the actions in the flow diagram is quite informal, you can easily determine the actual steps performed by the function. For instance, in order to check whether the requested page is the same as the page previously read, the function checks whether the values of the `ra->prev_page` field and of the `offset` parameter coincide (see Table 16-3 earlier).

When the process accesses the file for the first time and the first requested page is the page at offset zero in the file, the function assumes that the process will perform sequential accesses. Thus, the function creates a new current window starting from the first page. The length of the initial current window—always a power of two—is somewhat related to the number of pages requested by the process in the first read operation: the higher the number of requested pages, the larger the current window, up to the maximum value stored in the `ra->ra_pages` field. Conversely, when the process accesses the file for the first time but the first requested page is not at offset zero, the function assumes that the process will not perform sequential accesses. Thus, the function temporarily disables read-ahead (`ra->size` field is set to -1). However, a new current window is created when the function recognizes a sequential access while read-ahead is temporarily disabled.

If the ahead window does not already exist, it is created as soon as the function recognizes that the process has performed a sequential access in the current window. The ahead window always starts from the page following the last page of the current window. Its length, however, is related to the length of the current window as follows: if the `RA_FLAG_MISS` flag is set, the length of the ahead window is the length of the current window minus 2, or four pages if the result is less than four; otherwise, the length of the ahead window is either four times or two times the length of the current window. If the process continues to access the file in a sequential way, eventually the ahead window becomes the new current window, and a new ahead window is created. Thus, read-ahead is aggressively enhanced if the process reads the file sequentially.

As soon as the function recognizes a file access that is not sequential with respect to the previous one, the current and ahead windows are cleared (emptied) and the read-ahead is temporarily disabled. Read-ahead is restarted from scratch as soon as the process performs a read operation that is sequential with respect to the previous file access.

---

[*] Actually, if the read operation involves a number of pages larger than the maximum size of the read-ahead window, the `page_cache_readahead()` function is invoked several times. Thus, the `req_size` parameter might be smaller than the number of pages yet to be read to complete the read operation.
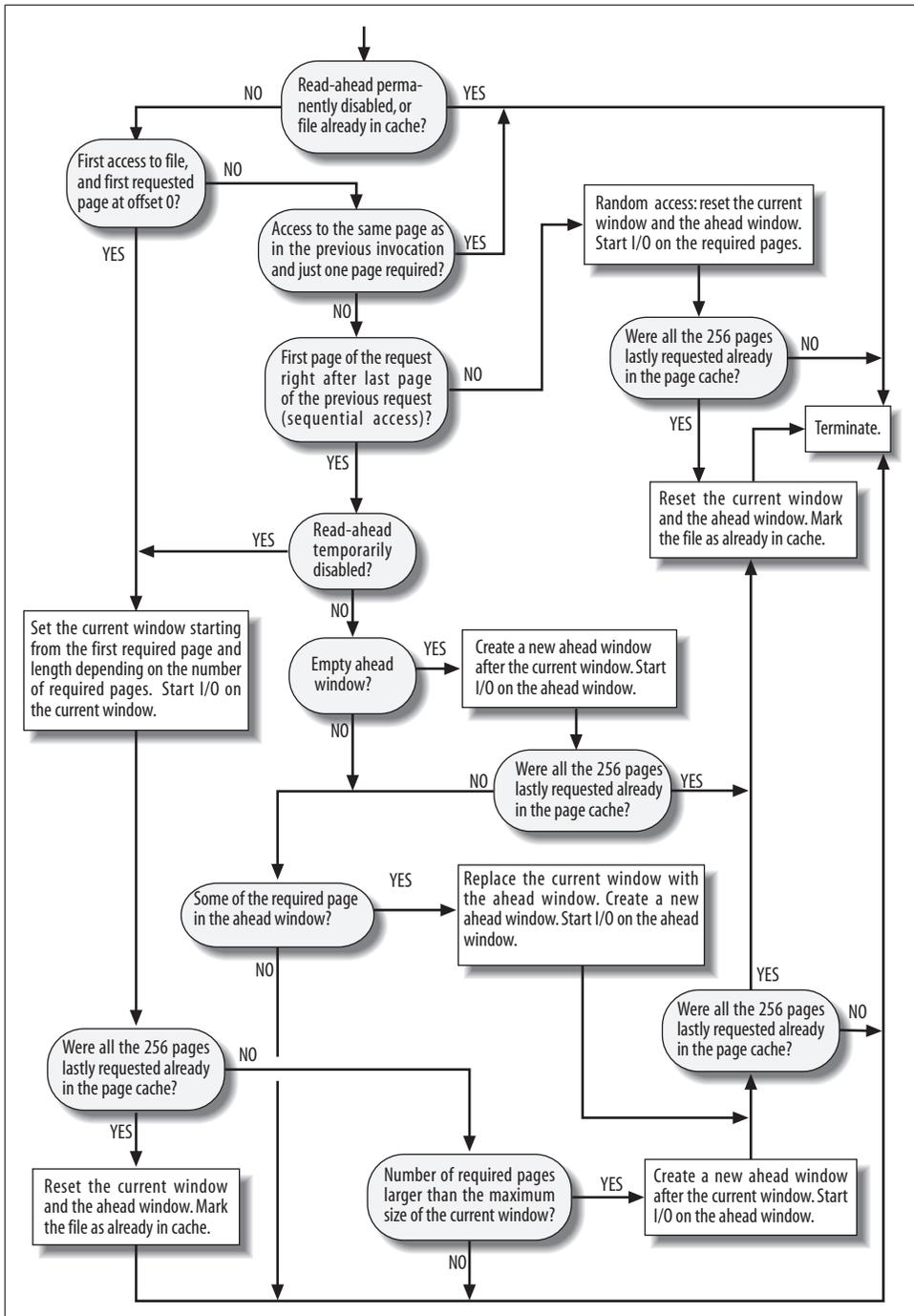
Figure 16-1. The flow diagram of the page_cache_readahead() function

Every time `page_cache_readahead()` creates a new window, it starts the read operations for the included pages. In order to read a chunk of pages, `page_cache_readahead()` invokes the `blockable_page_cache_readahead()` function. To reduce kernel overhead, the latter function adopts the following clever features:

- No reading is performed if the request queue that services the block device is read-congested (it does not make sense to increase congestion and block read-ahead).

- The page cache is checked against each page to be read; if the page is already in the page cache, it is simply skipped over.

- All the page frames needed by the read request are allocated at once before performing the read from disk. If not all page frames can be obtained, the read-ahead operation is performed only on the available pages. Again, there is little sense in deferring read-ahead until all page frames become available.

- Whenever possible, the read operations are submitted to the generic block layer by using multi-segment bio descriptors (see the section "Segments" in Chapter 14). This is done by the specialized `readpages` method of the `address_space` object, if defined; otherwise, it is done by repeatedly invoking the `readpage` method. The `readpage` method is described in the earlier section "Reading from a File" for the single-segment case only, but it is easy to adapt the description for the multi-segment case.

The handle_ra_miss() function

In some cases, the kernel must correct the read-ahead parameters, because the read-ahead strategy does not seem very effective. Let us consider the `do_generic_file_read()` function described in the section "Reading from a File" earlier in this chapter. The `page_cache_readahead()` function is invoked in step 4c. The flow diagram in Figure 16-1 depicts two cases: either the requested page is in the current window or in the ahead window, hence it should have been read in advance, or it is not, and the function invokes `blockable_page_cache_readahead()` to read it. In both cases, `do_generic_file_read()` should find the page in the page cache in step 4d. If it is not found, this means that the page frame reclaiming algorithm has removed the page from the cache. In this case, `do_generic_file_read()` invokes the `handle_ra_miss()` function, which tunes the read-ahead algorithm by setting the `RA_FLAG_MISS` flag and by clearing the `RA_FLAG_INCACHE` flag.

Writing to a File

Recall that the `write()` system call involves moving data from the User Mode address space of the calling process into the kernel data structures, and then to disk. The `write` method of the file object permits each filesystem type to define a specialized write operation. In Linux 2.6, the `write` method of each disk-based filesystem is a procedure that basically identifies the disk blocks involved in the write operation,

copies the data from the User Mode address space into some pages belonging to the page cache, and marks the buffers in those pages as dirty.

Many filesystems (including Ext2 or JFS) implement the `write` method of the file object by means of the `generic_file_write( )` function, which acts on the following parameters:

`file`

File object pointer

`buf`

Address in the User Mode address space where the characters to be written into the file must be fetched

`count`

Number of characters to be written

`ppos`

Address of a variable storing the file offset from which writing must start

The function performs the following steps:

1. Initializes a local variable of type `iovec` containing the address and length of the User Mode buffer (see also the description of the `generic_file_read( )` function in the section "Reading from a File" earlier in this chapter).

2. Determines the address `inode` of the inode object that corresponds to the file to be written (`file->f_mapping->host`) and acquires the semaphore `inode->i_sem`. Thanks to this semaphore, only one process at a time can issue a `write( )` system call on the file.

3. Invokes the `init_sync_kiocb` macro to initialize a local variable of type `kiocb`. As explained in the section "Reading from a File" earlier in this chapter, the macro sets the `ki_key` field to `KIOCB_SYNC_KEY` (synchronous I/O operation), the `ki_filp` field to `file`, and the `ki_obj` field to `current`.

4. Invokes `__generic_file_aio_write_nolock( )` (see below) to mark the affected pages as dirty, passing the address of the local variables of type `iovec` and `kiocb`, the number of segments for the User Mode buffer—only one in this case—and the parameter `ppos`.

5. Releases the `inode->i_sem` semaphore.

6. Checks the `O_SYNC` flag of the file, the `S_SYNC` flag of the inode, and the `MS_SYNCHRONOUS` flag of the superblock; if at least one of them is set, it invokes the `sync_page_range( )` function to force the kernel to flush all pages in the page cache that have been touched in step 4, blocking the current process until the I/O data transfers terminate. In turn, `sync_page_range( )` executes either the `writepages` method of the `address_space` object, if defined, or the `mpage_writepages( )` function (see the section "Writing Dirty Pages to Disk" later in this chapter) to start the I/O transfers for the dirty pages; then, it invokes `generic_osync_inode( )` to flush to disk the inode and the associated buffers, and finally

invokes wait_on_page_bit() to suspend the current process until all PG_
writeback bits of the flushed pages are cleared.

7. Returns the code returned by __generic_file_aio_write_nolock(), usually the
   number of bytes effectively written.

The __generic_file_aio_write_nolock() function receives four parameters: the
address iocb of a kiocb descriptor, the address iov of an array of iovec descriptors,
the length of this array, and the address ppos of a variable that stores the file's cur-
rent pointer. When invoked by generic_file_write(), the array of iovec descriptors
is composed of just one element describing the User Mode buffer that contains the
data to be written.[*]

We now explain the actions of the __generic_file_aio_write_nolock() function; for
the sake of simplicity, we restrict the description to the most common case: a com-
mon mode operation raised by a write() system call on a page-cached file. Later in
this chapter we describe how this function behaves in other cases. As usual, we do
not discuss how errors and anomalous conditions are handled.

The function executes the following steps:

1. Invokes access_ok() to verify that the User Mode buffer described by the iovec
   descriptor is valid (the starting address and length have been received from the
   sys_write() service routine, thus they must be checked before using them; see
   the section "Verifying the Parameters" in Chapter 10). If the parameters are not
   valid, it returns the -EFAULT error code.

2. Determines the address inode of the inode object that corresponds to the file to
   be written (file->f_mapping->host). Remember that if the file is a block device
   file, this is an inode in the *bdev* special filesystem (see Chapter 14).

3. Sets current->backing_dev_info to the address of the backing_dev_info descrip-
   tor of the file (file->f_mapping->backing_dev_info). Essentially, this setting
   allows the current process to write back the dirty pages owned by file->f_
   mapping even if the corresponding request queue is congested; see Chapter 17.

4. If the O_APPEND flag of file->flags is on and the file is regular (not a block device
   file), it sets *ppos to the end of the file so that all new data is appended to it.

5. Performs several checks on the size of the file. For instance, the write operation
   must not enlarge a regular file so much as to exceed the per-user limit stored in
   current->signal->rlim[RLIMIT_FSIZE] (see the section "Process Resource Lim-
   its" in Chapter 3) and the filesystem limit stored in inode->i_sb->s_maxbytes.

---

[*] A variant of the write() system call—named writev()—allows an application to define multiple User Mode
buffers from which the kernel fetches the data to be written on the file; the generic_file_aio_write_nolock()
function handles this case too. In the following pages, we will assume that the data will be fetched from just
one User Mode buffer; however, guessing the additional steps to be performed when using multiple buffers is
straightforward.

Moreover, if the file is not a "large file" (flag `O_LARGEFILE` of `file->f_flags` cleared), its size cannot exceed 2 GB. If any of these constraints is not enforced, it reduces the number of bytes to be written.

6. If set, it clears the `suid` flag of the file; also clears the `sgid` flag if the file is executable (see the section "Access Rights and File Mode" in Chapter 1). We don't want users to be able to modify *setuid* files.

7. Stores the current time of day in the `inode->mtime` field (the time of last file write operation) and in the `inode->ctime` field (the time of last inode change), and marks the inode object as dirty.

8. Starts a cycle to update all the pages of the file involved in the write operation. During each iteration, it performs the following substeps:

    a. Invokes `find_lock_page()` to search the page in the page cache (see the section "Page Cache Handling Functions" in Chapter 15). If this function finds the page, it increases its usage counter and sets its `PG_locked` flag.

    b. If the page is not in the page cache, it allocates a new page frame and invokes `add_to_page_cache()` to insert the page into the page cache; as explained in the section "Page Cache Handling Functions" in Chapter 15, this function also increases the usage counter and sets the `PG_locked` flag. Moreover, the function inserts the new page into the inactive list of the memory zone (see Chapter 17).

    c. Invokes the `prepare_write` method of the `address_space` object of the inode (`file->f_mapping`). The corresponding function takes care of allocating and initializing buffer heads for the page. We'll discuss in subsequent sections what this function does for regular files and block device files.

    d. If the buffer is in high memory, it establishes a kernel mapping of the User Mode buffer (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8). Then, it invokes `__copy_from_user()` to copy the characters from the User Mode buffer to the page, and releases the kernel mapping.

    e. Invokes the `commit_write` method of the `address_space` object of the inode (`file->f_mapping`). The corresponding function marks the underlying buffers as dirty so they are written to disk later. We discuss what this function does for regular files and block device files in the next two sections.

    f. Invokes `unlock_page()` to clear the `PG_locked` flag and wake up any process that is waiting for the page.

    g. Invokes `mark_page_accessed()` to update the page status for the memory reclaiming algorithm (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).

    h. Decreases the page usage counter to undo the increment in step 8a or 8b.

    i. In this iteration, yet another page has been dirtied: it checks whether the ratio of dirty pages in the page cache has risen above a fixed threshold (usually,

40% of the pages in the system); if so, it invokes `writeback_inodes()` to start flushing a few tens of pages to disk (see the section "Looking for Dirty Pages To Be Flushed" in Chapter 15).

   j. Invokes `cond_resched()` to check the `TIF_NEED_RESCHED` flag of the current process and, if the flag is set, to invoke the `schedule()` function.

9. Now all pages of the file involved in the write operation have been handled. Updates the value of *ppos to point right after the last character written.

10. Sets `current->backing_dev_info` to `NULL` (see step 3).

11. Terminates by returning the number of bytes effectively written.

The prepare_write and commit_write methods for regular files

The `prepare_write` and `commit_write` methods of the `address_space` object specialize the generic write operation implemented by `generic_file_write()` for regular files and block device files. Both of them are invoked once for every page of the file that is affected by the write operation.

Each disk-based filesystem defines its own `prepare_write` method. As with read operations, this method is simply a wrapper for a common function. For instance, the Ext2 filesystem usually implements the `prepare_write` method by means of the following function:

```
int ext2_prepare_write(struct file *file, struct page *page,
                       unsigned from, unsigned to)
{
    return block_prepare_write(page, from, to, ext2_get_block);
}
```

The `ext2_get_block()` function was already mentioned in the earlier section "Reading from a File"; it translates the block number relative to the file into a logical block number, which represents the position of the data on the physical block device.

The `block_prepare_write()` function takes care of preparing the buffers and the buffer heads of the file's page by performing essentially the following steps:

1. Checks if the page is a buffer page (flag `PG_Private` set); if this flag is cleared, invokes `create_empty_buffers()` to allocate buffer heads for all buffers included in the page (see the section "Buffer Pages" in Chapter 15).

2. For each buffer head relative to a buffer included in the page and affected by the write operation, the following is performed:

   a. Resets the `BH_New` flag, if it is set (see below).

   b. If the `BH_Mapped` flag is not set, the function performs the following substeps:

      1. Invokes the filesystem-dependent function whose address `get_block` was passed as a parameter. This function looks in the on-disk data structures of the filesystem and finds the logical block number of the buffer (relative to the beginning of the disk partition rather than the beginning

of the regular file). The filesystem-dependent function stores this number in the b_blocknr field of the corresponding buffer head and sets its BH_Mapped flag. The get_block function could allocate a new physical block for the file (for instance, if the accessed block falls inside a "hole" of the regular file; see the section "File Holes" in Chapter 18). In this case, it sets the BH_New flag.

2. Checks the value of the BH_New flag; if it is set, invokes unmap_underlying_metadata() to check whether some block device buffer page in the page cache includes a buffer referencing the same block on disk.[*] This function essentially invokes __find_get_block() to look up the old block in the page cache (see the section "Searching Blocks in the Page Cache" in Chapter 15). If such a block is found, the function clears its BH_Dirty flag and waits until any I/O data transfer on that buffer completes. Moreover, if the write operation does not rewrite the whole buffer in the page, it fills the unwritten portion with zero's. Then it considers the next buffer in the page.

c. If the write operation does not rewrite the whole buffer and its BH_Delay and BH_Uptodate flags are not set (that is, the block has been allocated in the on-disk filesystem data structures and the buffer in RAM does not contain a valid image of the data), the function invokes ll_rw_block() on the block to read its content from disk (see the section "Submitting Buffer Heads to the Generic Block Layer" in Chapter 15).

3. Blocks the current process until all read operations triggered in step 2c have been completed.

4. Returns 0.

Once the prepare_write method returns, the generic_file_write() function updates the page with the data stored in the User Mode address space. Next, it invokes the commit_write method of the address_space object. This method is implemented by the generic_commit_write() function for almost all disk-based non-journaling filesystems.

The generic_commit_write() function performs essentially the following steps:

1. Invokes the __block_commit_write() function. In turn, this function does the following:

a. Considers all buffers in the page that are affected by the write operation; for each of them, sets the BH_Uptodate and BH_Dirty flags of the corresponding buffer head.

---

[*] Although unlikely, this case might happen if a user writes blocks directly on the block device file, thus bypassing the filesystem.

b. Marks the corresponding inode as dirty. As seen in the section "Looking for Dirty Pages To Be Flushed" in Chapter 15, this activity may require adding the inode to the list of dirty inodes of the superblock.

c. If all buffers in the buffer page are now up-to-date, it sets the PG_uptodate flag of the page.

d. Sets the PG_dirty flag of the page, and tags the page as dirty in its radix tree (see the section "The Radix Tree" in Chapter 15).

2. Checks whether the write operation enlarged the file. In this case, the function updates the i_size field of the file's inode.

3. Returns 0.

The prepare_write and commit_write methods for block device files

Write operations into block device files are very similar to the corresponding operations on regular files. In fact, the prepare_write method of the address_space object of block device files is usually implemented by the following function:

```
int blkdev_prepare_write(struct file *file, struct page *page,
                         unsigned from, unsigned to)
{
    return block_prepare_write(page, from, to, blkdev_get_block);
}
```

As you can see, the function is simply a wrapper to the block_prepare_write() function already discussed in the previous section. The only difference, of course, is in the second parameter, which points to the function that must translate the file block number relative to the beginning of the file to a logical block number relative to the beginning of the block device. Remember that for block device files, the two numbers coincide. (See the earlier section "Reading from a File" for a discussion of the blkdev_get_block() function.)

The commit_write method for block device files is implemented by the following simple wrapper function:

```
int blkdev_commit_write(struct file *file, struct page *page,
                        unsigned from, unsigned to)
{
    return block_commit_write(page, from, to);
}
```

As you can see, the commit_write method for block device files does essentially the same things as the commit_write method for regular files (we described the block_commit_write() function in the previous section). The only difference is that the method does not check whether the write operation has enlarged the file; you simply cannot enlarge a block device file by appending characters to its last position.

Writing Dirty Pages to Disk

The net effect of the `write()` system call consists of modifying the contents of some pages in the page cache—optionally allocating the pages and adding them to the page cache if they were not already present. In some cases (for instance, if the file has been opened with the `O_SYNC` flag), the I/O data transfers start immediately (see step 6 of `generic_file_write()` in the section "Writing to a File" earlier in this chapter). Usually, however, the I/O data transfer is delayed, as explained in the section "Writing Dirty Pages to Disk" in Chapter 15.

When the kernel wants to effectively start the I/O data transfer, it ends up invoking the `writepages` method of the file's `address_space` object, which searches for dirty pages in the radix-tree and flushes them to disk. For instance, the Ext2 filesystem implements the `writepages` method by means of the following function:

```
int ext2_writepages(struct address_space *mapping,
                    struct writeback_control *wbc)
{
    return mpage_writepages(mapping, wbc, ext2_get_block);
}
```

As you can see, this function is a simple wrapper for the general-purpose `mpage_writepages()` function; as a matter of fact, if a filesystem does not define the `writepages` method, the kernel invokes directly `mpage_writepages()` passing `NULL` as third argument. The `ext2_get_block()` function was already mentioned in the earlier section "Reading from a File;" it is the filesystem-dependent function that translates a file block number into a logical block number.

The `writeback_control` data structure is a descriptor that controls how the writeback operation has to be performed; we have already described it in the section "Looking for Dirty Pages To Be Flushed" in Chapter 15.

The `mpage_writepages()` function essentially performs the following actions:

1. If the request queue is write-congested and the process does not want to block, it returns without writing any page to disk.

2. Determines the file's initial page to be considered. If the `writeback_control` descriptor specifies the initial position in the file, the function translates it into a page index. Otherwise, if the `writeback_control` descriptor specifies that the process does not want to wait for the I/O data transfer to complete, it sets the initial page index to the value stored in `mapping->writeback_index` (that is, scanning begins from the last page considered in the previous writeback operation). Finally, if the process must wait until I/O data transfers complete, scanning starts from the first page of the file.

3. Invokes `find_get_pages_tag()` to look up the descriptor of the dirty pages in the page cache (see the section "The Tags of the Radix Tree" in Chapter 15).

4. For each page descriptor retrieved in the previous step, the function performs the following steps:

   a. Invokes `lock_page()` to lock up the page.

   b. Checks that the page is still valid and in the page cache (because another kernel control path could have acted upon the page between steps 3 and 4a).

   c. Checks the `PG_writeback` flag of the page. If it is set, the page is already being flushed to disk. If the process must wait for the I/O data transfer to complete, it invokes `wait_on_page_bit()` to block the current process until the `PG_writeback` flag is cleared; when this function terminates, any previously ongoing writeback operation is terminated. Otherwise, if the process does not want to wait, it checks the `PG_dirty` flag: if it is now cleared, the ongoing writeback will take care of the page, thus unlocks it and jumps back to step 4a to continue with the next page.

   d. If the `get_block` parameter is NULL (no `writepages` method defined), it invokes the `mapping->writepage` method of the `address_space` object of the file to flush the page to disk. Otherwise, if the `get_block` parameter is not NULL, it invokes the `mpage_writepage()` function. See step 8 for details.

5. Invokes `cond_resched()` to check the `TIF_NEED_RESCHED` flag of the current process and, if the flag is set, to invoke the `schedule()` function.

6. If the function has not scanned all pages in the given range, or if the number of pages effectively written to disk is smaller than the value originally specified in the `writeback_control` descriptor, it jumps back to step 3.

7. If the `writeback_control` descriptor does not specify the initial position in the file, it sets the `mapping->writeback_index` field with the index of the last scanned page.

8. If the `mpage_writepage()` function has been invoked in step 4d, and if that function returned the address of a bio descriptor, it invokes `mpage_bio_submit()` (see below).

A typical filesystem such as Ext2 implements the `writepage` method as a wrapper for the general-purpose `block_write_full_page()` function, passing to it the address of the filesystem-dependent `get_block` function. In turn, the `block_write_full_page()` function is similar to `block_read_full_page()` described in the section "Reading from a File" earlier in this chapter: it allocates buffer heads for the page (if the page was not already a buffer page), and invokes the `submit_bh()` function on each of them, specifying the WRITE operation. As far as block device files are concerned, they implement the `writepage` method by using `blkdev_writepage()`, which is a wrapper for `block_write_full_page()`.

Many non-journaling filesystems rely on the `mpage_writepage()` function rather than on the custom `writepage` method. This can improve performance because the `mpage_writepage()` function tries to submit the I/O transfers by collecting as many pages as

possible in the same bio descriptor; in turn, this allows the block device drivers to exploit the scatter-gather DMA capabilities of the modern hard disk controllers.

To make a long story short, the `mpage_writepage()` function checks whether the page to be written contains blocks that are not adjacent to disk, or whether the page includes a file hole, or whether some block on the page is not dirty or not up-to-date. If at least one of these conditions holds, the function falls back on the filesystem-dependent `writepage` method, as above. Otherwise, the function adds the page as a segment of a bio descriptor. The address of the bio descriptor is passed as parameter to the function; if it is `NULL`, `mpage_writepage()` initializes a new bio descriptor and returns its address to the calling function, which in turn passes it back in the future invocations of `mpage_writepage()`. In this way, several pages can be added to the same bio. If a page is not adjacent to the last added page in the bio, `mpage_writepage()` invokes `mpage_bio_submit()` to start the I/O data transfer on the bio, and allocates a new bio for the page.

The `mpage_bio_submit()` function sets the `bi_end_io` method of the bio to the address of `mpage_end_io_write()`, then invokes `submit_bio()` to start the transfer (see the section "Submitting Buffer Heads to the Generic Block Layer" in Chapter 15). Once the data transfer successfully terminates, the completion function `mpage_end_io_write()` wakes up any process waiting for the page transfer to complete, and destroys the bio descriptor.

Memory Mapping

As already mentioned in the section "Memory Regions" in Chapter 9, a memory region can be associated with some portion of either a regular file in a disk-based filesystem or a block device file. This means that an access to a byte within a page of the memory region is translated by the kernel into an operation on the corresponding byte of the file. This technique is called *memory mapping*.

Two kinds of memory mapping exist:

Shared
> Each write operation on the pages of the memory region changes the file on disk; moreover, if a process writes into a page of a shared memory mapping, the changes are visible to all other processes that map the same file.

Private
> Meant to be used when the process creates the mapping just to read the file, not to write it. For this purpose, private mapping is more efficient than shared mapping. But each write operation on a privately mapped page will cause it to stop mapping the page in the file. Thus, a write does not change the file on disk, nor is the change visible to any other processes that access the same file. However, pages of a private memory mapping that have not been modified by the process are affected by file updates performed by other processes.

A process can create a new memory mapping by issuing an mmap( ) system call (see the section "Creating a Memory Mapping" later in this chapter). Programmers must specify either the MAP_SHARED flag or the MAP_PRIVATE flag as a parameter of the system call; as you can easily guess, in the former case the mapping is shared, while in the latter it is private. Once the mapping is created, the process can read the data stored in the file by simply reading from the memory locations of the new memory region. If the memory mapping is shared, the process can also modify the corresponding file by simply writing into the same memory locations. To destroy or shrink a memory mapping, the process may use the munmap( ) system call (see the later section "Destroying a Memory Mapping").

As a general rule, if a memory mapping is shared, the corresponding memory region has the VM_SHARED flag set; if it is private, the VM_SHARED flag is cleared. As we'll see later, an exception to this rule exists for read-only shared memory mappings.

Memory Mapping Data Structures

A memory mapping is represented by a combination of the following data structures:

- The inode object associated with the mapped file
- The address_space object of the mapped file
- A file object for each different mapping performed on the file by different processes
- A vm_area_struct descriptor for each different mapping on the file
- A page descriptor for each page frame assigned to a memory region that maps the file

Figure 16-2 illustrates how the data structures are linked. On the left side of the image we show the inode, which identifies the file. The i_mapping field of each inode object points to the address_space object of the file. In turn, the page_tree field of each address_space object points to the radix tree of pages belonging to the address space (see the section "The Radix Tree" in Chapter 15), while the i_mmap field points to a second tree called the radix priority search tree (PST) of memory regions belonging to the address space. The main use of PST is for performing "reverse mapping," that is, for identifying quickly all processes that share a given page. We'll cover in detail PSTs in the next chapter, because they are used for page frame reclaiming. The link between file objects relative to the same file and the inode is established by means of the f_mapping field.

Each memory region descriptor has a vm_file field that links it to the file object of the mapped file (if that field is null, the memory region is not used in a memory mapping). The position of the first mapped location is stored into the vm_pgoff field of the memory region descriptor; it represents the file offset as a number of page-size units. The length of the mapped file portion is simply the length of the memory region, which can be computed from the vm_start and vm_end fields.

mapping mapping mapping

address_space
object

page_tree

struct
page

Radix
tree

struct
page

struct
page

index index index

struct
inode

File
image

i_mapping

page
frame

page
frame

page
frame

vm_pgoff

Memory regions

vm_pgoff

i_mmap

struct
vm_area_struct

Radix priority
search tree

struct
vm_area_struct

vm_file vm_file

f_mapping

struct
file

f_mapping
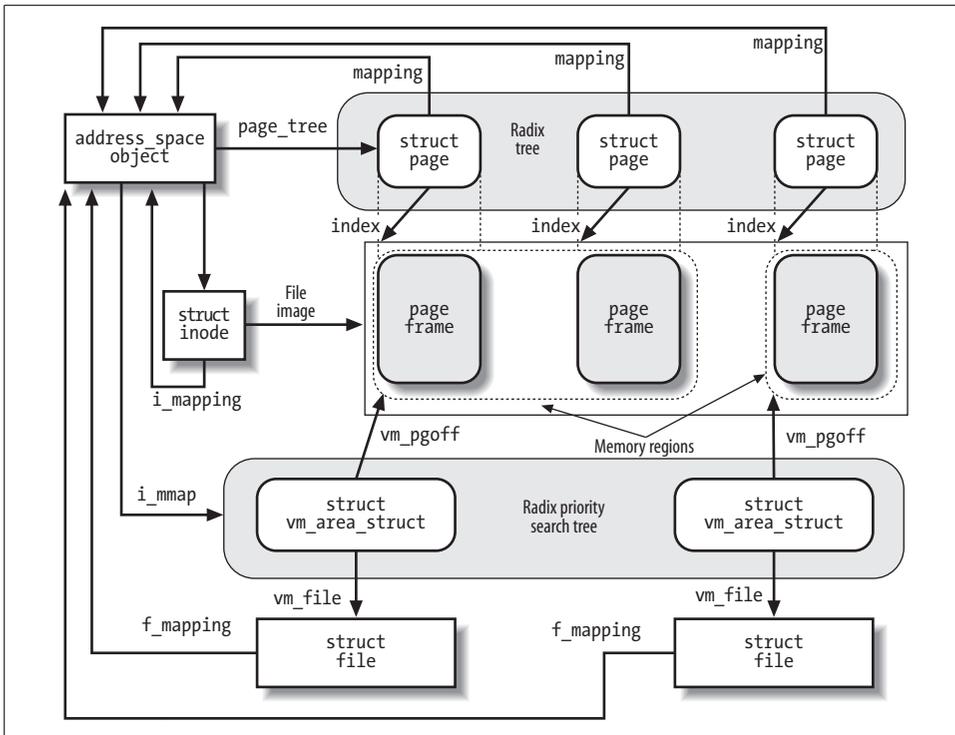
struct
file

Figure 16-2. Data structures for file memory mapping

Pages of shared memory mappings are always included in the page cache; pages of private memory mappings are included in the page cache as long as they are unmodified. When a process tries to modify a page of a private memory mapping, the kernel duplicates the page frame and replaces the original page frame with the duplicate in the process Page Table; this is one of the applications of the Copy On Write mechanism that we discussed in Chapter 8. The original page frame still remains in the page cache, although it no longer belongs to the memory mapping since it is replaced by the duplicate. In turn, the duplicate is not inserted into the page cache because it no longer contains valid data representing the file on disk.

Figure 16-2 also shows a few page descriptors of pages included in the page cache that refer to the memory-mapped file. Notice that the first memory region in the figure is three pages long, but only two page frames are allocated for it; presumably, the process owning the memory region has never accessed the third page.

The kernel offers several hooks to customize the memory mapping mechanism for every different filesystem. The core of memory mapping implementation is delegated to a file object's method named mmap. For most disk-based filesystems and for block device files, this method is implemented by a general function called generic_file_mmap( ), which is described in the next section.

File memory mapping depends on the demand paging mechanism described in the section "Demand Paging" in Chapter 9. In fact, a newly established memory mapping is a memory region that doesn't include any page; as the process references an address inside the region, a Page Fault occurs and the Page Fault handler checks whether the `nopage` method of the memory region is defined. If `nopage` is not defined, the memory region doesn't map a file on disk; otherwise, it does, and the method takes care of reading the page by accessing the block device. Almost all disk-based filesystems and block device files implement the `nopage` method by means of the `filemap_nopage()` function.

Creating a Memory Mapping

To create a new memory mapping, a process issues an `mmap()` system call, passing the following parameters to it:

- A file descriptor identifying the file to be mapped.
- An offset inside the file specifying the first character of the file portion to be mapped.
- The length of the file portion to be mapped.
- A set of flags. The process must explicitly set either the `MAP_SHARED` flag or the `MAP_PRIVATE` flag to specify the kind of memory mapping requested.[*]
- A set of permissions specifying one or more types of access to the memory region: read access (`PROT_READ`), write access (`PROT_WRITE`), or execution access (`PROT_EXEC`).
- An optional linear address, which is taken by the kernel as a hint of where the new memory region should start. If the `MAP_FIXED` flag is specified and the kernel cannot allocate the new memory region starting from the specified linear address, the system call fails.

The `mmap()` system call returns the linear address of the first location in the new memory region. For compatibility reasons, in the 80×86 architecture, the kernel reserves two entries in the system call table for `mmap()`: one at index 90 and the other at index 192. The former entry corresponds to the `old_mmap()` service routine (used by older C libraries), while the latter one corresponds to the `sys_mmap2()` service routine (used by recent C libraries). The two service routines differ only in how the six parameters of the system call are passed. Both of them end up invoking the `do_mmap_pgoff()` function described in the section "Allocating a Linear Address Interval" in

---

[*] The process could also set the `MAP_ANONYMOUS` flag to specify that the new memory region is anonymous—that is, not associated with any disk-based file (see the section "Demand Paging" in Chapter 9). A process can also create a memory region that is both `MAP_SHARED` and `MAP_ANONYMOUS`: in this case, the region maps a special file in the *tmpfs* filesystem (see the section "IPC Shared Memory" in Chapter 19), which can be accessed by all the process's descendants.

Chapter 9. We now complete that description by detailing the steps performed only when creating a memory region that maps a file. We thus describe the case where the file parameter (pointer to a file object) of do_mmap_pgoff() is non-null. For the sake of clarity, we refer to the enumeration used to describe do_mmap_pgoff() and point out the additional steps performed under the new condition.

Step 1

Checks whether the mmap file operation for the file to be mapped is defined; if not, it returns an error code. A NULL value for mmap in the file operation table indicates that the corresponding file cannot be mapped (for instance, because it is a directory).

Step 2

The get_unmapped_area() function invokes the get_unmapped_area method of the file object, if it is defined, so as to allocate an interval of linear addresses suitable for the memory mapping of the file. The disk-based filesystems do not define this method; in this case, as explained in the section "Memory Region Handling" in Chapter 9, the get_unmapped_area() function ends up invoking the get_unmapped_area method of the memory descriptor.

Step 3

In addition to the usual consistency checks, it compares the kind of memory mapping requested (stored in the flags parameter of the mmap() system call) and the flags specified when the file was opened (stored in the file->f_mode field). In particular:

- If a shared writable memory mapping is required, it checks that the file was opened for writing and that it was not opened in append mode (O_APPEND flag of the open() system call).

- If a shared memory mapping is required, it checks that there is no mandatory lock on the file (see the section "File Locking" in Chapter 12).

- For every kind of memory mapping, it checks that the file was opened for reading.

If any of these conditions is not fulfilled, an error code is returned.

Moreover, when initializing the value of the vm_flags field of the new memory region descriptor, it sets the VM_READ, VM_WRITE, VM_EXEC, VM_SHARED, VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC, and VM_MAYSHARE flags according to the access rights of the file and the kind of requested memory mapping (see the section "Memory Region Access Rights" in Chapter 9). As an optimization, the VM_SHARED and VM_MAYWRITE flags are cleared for nonwritable shared memory mapping. This can be done because the process is not allowed to write into the pages of the memory region, so the mapping is treated the same as a private mapping; however, the kernel actually allows other processes that share the file to read the pages in this memory region.

Step 10

Initializes the `vm_file` field of the memory region descriptor with the address of the file object and increases the file's usage counter. Invokes the `mmap` method for the file being mapped, passing as parameters the address of the file object and the address of the memory region descriptor. For most filesystems, this method is implemented by the `generic_file_mmap()` function, which performs the following operations:

a. Stores the current time in the `i_atime` field of the file's inode and marks the inode as dirty.

b. Initializes the `vm_ops` field of the memory region descriptor with the address of the `generic_file_vm_ops` table. All methods in this table are null, except the `nopage` method, which is implemented by the `filemap_nopage()` function, and the `populate` method, which is implemented by the `filemap_populate()` function (see "Non-Linear Memory Mappings" later in this chapter).

Step 11

Increases the `i_writecount` field of the file's inode, that is, the usage counter for writing processes.

Destroying a Memory Mapping

When a process is ready to destroy a memory mapping, it invokes `munmap()`; this system call can also be used to reduce the size of each kind of memory region. The parameters used are:

- The address of the first location in the linear address interval to be removed.
- The length of the linear address interval to be removed.

The `sys_munmap()` service routine of the system call essentially invokes the `do_munmap()` function already described in the section "Releasing a Linear Address Interval" in Chapter 9. Notice that there is no need to flush to disk the contents of the pages included in a writable shared memory mapping to be destroyed. In fact, these pages continue to act as a disk cache because they are still included in the page cache.

Demand Paging for Memory Mapping

For reasons of efficiency, page frames are not assigned to a memory mapping right after it has been created, but at the last possible moment—that is, when the process attempts to address one of its pages, thus causing a Page Fault exception.

We saw in the section "Page Fault Exception Handler" in Chapter 9 how the kernel verifies whether the faulty address is included in some memory region of the process; if so, the kernel checks the Page Table entry corresponding to the faulty address

and invokes the do_no_page( ) function if the entry is null (see the section "Demand Paging" in Chapter 9).

The do_no_page( ) function performs all the operations that are common to all types of demand paging, such as allocating a page frame and updating the Page Tables. It also checks whether the nopage method of the memory region involved is defined. In the section "Demand Paging" in Chapter 9, we described the case in which the method is undefined (anonymous memory region); now we complete the description by discussing the main actions performed by the function when the method is defined:

1. Invokes the nopage method, which returns the address of a page frame that contains the requested page.

2. If the process is trying to write into the page and the memory mapping is private, it avoids a future Copy On Write fault by making a copy of the page just read and inserting it into the inactive list of pages (see Chapter 17). If the private memory mapping region does not already have a slave anonymous memory region that includes the new page, it either adds a new slave anonymous memory region or extends an existing one (see the section "Memory Regions" in Chapter 9). In the following steps, the function uses the new page instead of the page returned by the nopage method, so that the latter is not modified by the User Mode process.

3. If some other process has truncated or invalidated the page (the truncate_count field of the address_space descriptor is used for this kind of check), the function retries getting the page by jumping back to step 1.

4. Increases the rss field of the process memory descriptor to indicate that a new page frame has been assigned to the process.

5. Sets up the Page Table entry corresponding to the faulty address with the address of the page frame and the page access rights included in the memory region vm_page_prot field.

6. If the process is trying to write into the page, it forces the Read/Write and Dirty bits of the Page Table entry to 1. In this case, either the page frame is exclusively assigned to the process, or the page is shared; in both cases, writing to it should be allowed.

The core of the demand paging algorithm consists of the memory region's nopage method. Generally speaking, it must return the address of a page frame that contains the page accessed by the process. Its implementation depends on the kind of memory region in which the page is included.

When handling memory regions that map files on disk, the nopage method must first search for the requested page in the page cache. If the page is not found, the method must read it from disk. Most filesystems implement the nopage method by means of the filemap_nopage( ) function, which receives three parameters:

area

Descriptor address of the memory region, including the required page

address

Linear address of the required page

type

Pointer to a variable in which the function writes the type of page fault detected by the function (`VM_FAULT_MAJOR` or `VM_FAULT_MINOR`)

The `filemap_nopage( )` function executes the following steps:

1. Gets the file object address `file` from the `area->vm_file` field. Derives the `address_space` object address from `file->f_mapping`. Derives the inode object address from the `host` field of the `address_space` object.

2. Uses the `vm_start` and `vm_pgoff` fields of `area` to determine the offset within the file of the data corresponding to the page starting from `address`.

3. Checks whether the file offset exceeds the file size. When this happens, it returns `NULL`, which means failure in allocating the new page, unless the Page Fault was caused by a debugger tracing another process through the `ptrace( )` system call. We are not going to discuss this special case.

4. If the `VM_RAND_READ` flag of the memory region is set (see below), we may assume that the process is reading the pages of the memory mapping in a random way. In this case, it ignores read-ahead by jumping to step 10.

5. If the `VM_SEQ_READ` flag of the memory region is set (see below), we may assume that the process is reading the pages of the memory mapping in a strictly sequential way. In this case, it invokes `page_cache_readahead( )` to perform read-ahead starting from the faulty page (see the section "Read-Ahead of Files" earlier in this chapter).

6. Invokes `find_get_page( )` to look in the page cache for the page identified by the `address_space` object and the file offset. If the page is found, it jumps to step 11.

7. If the function has reached this point, the page has not been found in the page cache. Checks the `VM_SEQ_READ` flag of the memory region:

    - If the flag is set, the kernel is aggressively reading in advance the pages of the memory region, hence the read-ahead algorithm has failed: it invokes `handle_ra_miss( )` to tune up the read-ahead parameters (see the section "Read-Ahead of Files" earlier in this chapter), then jumps to step 10.

    - Otherwise, if the flag is clear, it increases by one the `mmap_miss` counter in the `file_ra_state` descriptor of the file. If the number of misses is much larger than the number of hits (stored in the `mmap_hit` counter), it ignores read-ahead by jumping to step 10.

8. If read-ahead is not permanently disabled (`ra_pages` field in the `file_ra_state` descriptor greater than zero), it invokes `do_page_cache_readahead( )` to read a set of pages surrounding the requested page.

9. Invokes `find_get_page()` to check whether the requested page is in the page cache; if it is there, jumps to step 11.

10. Invokes `page_cache_read()`. This function checks whether the requested page is already in the page cache and, if it is not there, allocates a new page frame, adds it to the page cache, and executes the `mapping->a_ops->readpage` method to schedule an I/O operation that reads the page's contents from disk.

11. Invokes the `grab_swap_token()` function to possibly assign the swap token to the current process (see the section "The Swap Token" in Chapter 17).

12. The requested page is now in the page cache. Increases by one the `mmap_hit` counter of the `file_ra_state` descriptor of the file.

13. If the page is not up-to-date (`PG_uptodate` flag clear), it invokes `lock_page()` to lock up the page, executes the `mapping->a_ops->readpage` method to trigger the I/O data transfer, and invokes `wait_on_page_bit()` to sleep until the page is unlocked—that is, until the data transfer completes.

14. Invokes `mark_page_accessed()` to mark the requested page as accessed (see next chapter).

15. If an up-to-date version of the page was found in the page cache, it sets *type to `VM_FAULT_MINOR`; otherwise sets it to `VM_FAULT_MAJOR`.

16. Returns the address of the requested page.

A User Mode process can tailor the read-ahead behavior of the `filemap_nopage()` function by using the `madvise()` system call. The `MADV_RANDOM` command sets the `VM_RAND_READ` flag of the memory region to specify that the pages of the memory region will be accessed in random order; the `MADV_SEQUENTIAL` command sets the `VM_SEQ_READ` flag to specify that the pages will be accessed in strictly sequential order; finally, the `MADV_NORMAL` command resets both the `VM_RAND_READ` and `VM_SEQ_READ` flags to specify that the pages will be accessed in a unspecified order.

Flushing Dirty Memory Mapping Pages to Disk

The `msync()` system call can be used by a process to flush to disk dirty pages belonging to a shared memory mapping. It receives as its parameters the starting address of an interval of linear addresses, the length of the interval, and a set of flags that have the following meanings:

`MS_SYNC`
Asks the system call to suspend the process until the I/O operation completes. In this way, the calling process can assume that when the system call terminates, all pages of its memory mapping have been flushed to disk.

`MS_ASYNC` (*complement of* `MS_SYNC`)
Asks the system call to return immediately without suspending the calling process.

MS_INVALIDATE

> Asks the system call to invalidate other memory mappings of the same file (not really implemented, because useless in Linux).

The `sys_msync( )` service routine invokes `msync_interval( )` on each memory region included in the interval of linear addresses. In turn, the latter function performs the following operations:

1. If the `vm_file` field of the memory region descriptor is `NULL`, or if the `VM_SHARED` flag is clear, it returns 0 (the memory region is not a writable shared memory mapping of a file).

2. Invokes the `filemap_sync( )` function, which scans the Page Table entries corresponding to the linear address intervals included in the memory region. For each page found, it resets the `Dirty` flag in the corresponding page table entry and invokes `flush_tlb_page( )` to flush the corresponding translation lookaside buffers; then, it sets the `PG_dirty` flag in the page descriptor to mark the page as dirty.

3. If the `MS_ASYNC` flag is set, it returns. Therefore, the practical effect of the `MS_ASYNC` flag consists of setting the `PG_dirty` flags of the pages in the memory region; the system call does not actually start the I/O data transfers.

4. If the function has reached this point, the `MS_SYNC` flag is set, hence the function must flush the pages in the memory region to disk and put the current process to sleep until all I/O data transfers terminate. In order to do this, the function acquires the `i_sem` semaphore of the file's inode.

5. Invokes the `filemap_fdatawrite( )` function, which receives the address of the file's `address_space` object. This function essentially sets up a `writeback_control` descriptor with the `WB_SYNC_ALL` synchronization mode, and checks whether the address space has a built-in `writepages` method. If so, it invokes the corresponding function and returns. In the opposite case, it executes the `mpage_writepages( )` function. (See the section "Writing Dirty Pages to Disk" earlier in this chapter.)

6. Checks whether the `fsync` method of the file object is defined; if so, executes it. For regular files, this method usually limits itself to flushing the inode object of the file to disk. For block device files, however, the method invokes `sync_blockdev( )`, which activates the I/O data transfer of all dirty buffers of the device.

7. Executes the `filemap_fdatawait( )` function. We recall from the section "The Tags of the Radix Tree" in Chapter 15 that a radix tree in the page cache identifies all pages that are currently being written to disk by means of the `PAGECACHE_TAG_WRITEBACK` tag. The function quickly scans the portion of the radix tree that covers the given interval of linear addresses looking for pages having the `PG_writeback` flag set; for each such page, the function invokes `wait_on_page_bit( )` to sleep until the `PG_writeback` flag is cleared—that is, until the ongoing I/O data transfer on the page terminates.

8. Releases the `i_sem` semaphore of the file and returns.

Non-Linear Memory Mappings

The Linux 2.6 kernel offers yet another kind of access method for regular files: the *non-linear memory mappings*. Basically, a non-linear memory mapping is a file memory mapping as described previously, but its memory pages are not mapped to sequential pages on the file; rather, each memory page maps a random (arbitrary) page of file's data.

Of course, a User Mode application might achieve the same result by invoking the `mmap()` system call repeatedly, each time on a different 4096-byte-long portion of the file. However, this approach is not very efficient for non-linear mapping of large files, because each mapping page requires its own memory region.

In order to support non-linear memory mapping, the kernel makes use of a few additional data structures. First of all, the `VM_NONLINEAR` flag of the memory region descriptor specifies that the memory region contains a non-linear mapping. All descriptors of non-linear mapping memory regions for a given file are collected in a doubly linked circular list rooted at the `i_mmap_nonlinear` field of the `address_space` object.

To create a non-linear memory mapping, the User Mode application first creates a normal shared memory mapping with the `mmap()` system call. Then, the application remaps some of the pages in the memory mapping region by invoking `remap_file_pages()`. The `sys_remap_file_pages()` service routine of the system call receives four parameters:

start
: A linear address inside a shared file memory mapping region of the calling process

size
: Size of the remapped portion of the file in bytes

prot
: Unused (must be zero)

pgoff
: Page index of the initial file's page to be remapped

flags
: Flags controlling the non-linear memory mapping

The service routine remaps the portion of the file's data identified by the `pgoff` and `size` parameters starting from the `start` linear address. If either the memory region is not shared or it is not large enough to include all the pages requested for the mapping, the system call fails and an error code is returned. Essentially, the service routine inserts the memory region in the `i_mmap_nonlinear` list of the file and invokes the `populate` method of the memory region.

For all regular files, the populate method is implemented by the filemap_populate() function, which executes the following steps:

1. Checks whether the MAP_NONBLOCK flag in the flags parameter of the remap_file_pages() system call is clear; if so, it invokes do_page_cache_readahead() to read in advance the pages of the file to be remapped.

2. For each page to be remapped, performs the following substeps:

   a. Checks whether the page descriptor is already included in the page cache; if it is not there and the MAP_NONBLOCK flag is cleared, it reads the page from disk.

   b. If the page descriptor is in the page cache, it updates the Page Table entry of the corresponding linear address so that it points to the page frame, and updates the counter of pages in the memory region descriptor.

   c. Otherwise, if the page descriptor has not been found in the page cache, it stores the offset of the file's page in the 32 highest-order bits of the Page Table entry for the corresponding linear address; also, clears the Present bit of the Page Table entry and sets the Dirty bit.

As explained in the section "Demand Paging" in Chapter 9, when handling a demand-paging fault the handle_pte_fault() function checks the Present and Dirty bits in the Page Table entry; if they have the values corresponding to a non-linear memory mapping, handle_pte_fault() invokes the do_file_page() function, which extracts the index of the requested file's page from the high-order bits of the Page Table entry; then, do_file_page() invokes the populate method of the memory region to read the page from disk and update the Page Table entry itself.

Because the memory pages of a non-linear memory mapping are included in the page cache according to the page index relative to the beginning of the file—rather than the index relative to the beginning of the memory region—non-linear memory mappings are flushed to disk exactly like linear memory mappings (see the section "Flushing Dirty Memory Mapping Pages to Disk" earlier in this chapter).

Direct I/O Transfers

As we have seen, in Version 2.6 of Linux, there is no substantial difference between accessing a regular file through the filesystem, accessing it by referencing its blocks on the underlying block device file, or even establishing a file memory mapping. There are, however, some highly sophisticated programs (*self-caching applications*) that would like to have full control of the whole I/O data transfer mechanism. Consider, for example, high-performance database servers: most of them implement their own caching mechanisms that exploit the peculiar nature of the queries to the database. For these kinds of programs, the kernel page cache doesn't help; on the contrary, it is detrimental for the following reasons:

- Lots of page frames are wasted to duplicate disk data already in RAM (in the user-level disk cache).

- The read() and write() system calls are slowed down by the redundant instructions that handle the page cache and the read-ahead; ditto for the paging operations related to the file memory mappings.

- Rather than transferring the data directly between the disk and the user memory, the read() and write() system calls make two transfers: between the disk and a kernel buffer and between the kernel buffer and the user memory.

Because block hardware devices *must* be handled through interrupts and Direct Memory Access (DMA), and this can be done only in Kernel Mode, some sort of kernel support is definitely required to implement self-caching applications.

Linux offers a simple way to bypass the page cache: *direct I/O transfers*. In each I/O direct transfer, the kernel programs the disk controller to transfer the data directly from/to pages belonging to the User Mode address space of a self-caching application.

As we know, each data transfer proceeds asynchronously. While it is in progress, the kernel may switch the current process, the CPU may return to User Mode, the pages of the process that raised the data transfer might be swapped out, and so on. This works just fine for ordinary I/O data transfers because they involve pages of the disk caches. Disk caches are owned by the kernel, cannot be swapped out, and are visible to all processes in Kernel Mode.

On the other hand, direct I/O transfers should move data within pages that belong to the User Mode address space of a given process. The kernel must take care that these pages are accessible by every process in Kernel Mode and that they are not swapped out while the data transfer is in progress. Let us see how this is achieved.

When a self-caching application wishes to directly access a file, it opens the file specifying the O_DIRECT flag (see the section "The open() System Call" in Chapter 12). While servicing the open() system call, the dentry_open() function checks whether the direct_IO method is implemented for the address_space object of the file being opened, and returns an error code in the opposite case. The O_DIRECT flag can also be set for a file already opened by using the F_SETFL command of the fcntl() system call.

Let us consider first the case where the self-caching application issues a read() system call on a file opened with O_DIRECT. As mentioned in the section "Reading from a File" earlier in this chapter, the read file method is usually implemented by the generic_file_read() function, which initializes the iovec and kiocb descriptors and invokes __generic_file_aio_read(). The latter function verifies that the User Mode buffer described by the iovec descriptor is valid, then checks whether the O_DIRECT flag of the file is set. When invoked by a read() system call, the function executes a code fragment essentially equivalent to the following:

```
if (filp->f_flags & O_DIRECT) {
    if (count == 0 || *ppos > filp->f_mapping->host->i_size)
```

```
        return 0;
    retval = generic_file_direct_IO(READ, iocb, iov, *ppos, 1);
    if (retval > 0)
        *ppos += retval;
    file_accessed(filp);
    return retval;
}
```

The function checks the current values of the file pointer, the file size, and the number of requested characters, and then invokes the generic_file_direct_IO() function, passing to it the READ operation type, the iocb descriptor, the iovec descriptor, the current value of the file pointer, and the number of User Mode buffers specified in the io_vec descriptor (one). When generic_file_direct_IO() terminates, __generic_file_aio_read() updates the file pointer, sets the access timestamp on the file's inode, and returns.

Something similar happens when a write() system call is issued on a file having the O_DIRECT flag set. As mentioned in the section "Writing to a File" earlier in this chapter, the write method of the file ends up invoking generic_file_aio_write_nolock(): this function checks whether the O_DIRECT flag is set and, if so, invokes the generic_file_direct_IO() function, this time specifying the WRITE operation type.

The generic_file_direct_IO() function acts on the following parameters:

rw
    Type of operation: READ or WRITE

iocb
    Pointer to a kiocb descriptor (see Table 16-1)

iov
    Pointer to an array of iovec descriptors (see the section "Reading from a File" earlier in this chapter)

offset
    File offset

nr_segs
    Number of iovec descriptors in the iov array

The steps performed by generic_file_direct_IO() are the following:

1. Gets the address file of the file object from the ki_filp field of the kiocb descriptor, and the address mapping of the address_space object from the file->f_mapping field.

2. If the type of operation is WRITE and if one or more processes have created a memory mapping associated with a portion of the file, it invokes unmap_mapping_range() to unmap all pages of the file. This function also ensures that if any Page Table entry corresponding to a page to be unmapped has the Dirty bit set, then the corresponding page is marked as dirty in the page cache.

3. If the radix tree rooted at `mapping` is not empty (`mapping->nrpages` greater than zero), it invokes the `filemap_fdatawrite()` and `filemap_fdatawait()` functions to flush all dirty pages to disk and to wait until the I/O operations complete (see the section "Flushing Dirty Memory Mapping Pages to Disk" earlier in this chapter). (Even if the self-caching application is accessing the file directly, there could be other applications in the system that access the file through the page cache. To avoid data loss, the disk image is synchronized with the page cache before starting the direct I/O transfer.)

4. Invokes the `direct_IO` method of the `mapping` address space (see the following paragraphs).

5. If the operation type was `WRITE`, it invokes `invalidate_inode_pages2()` to scan all pages in the radix tree of `mapping` and to release them. The function also clears the User Mode Page Table entries that refer to those pages.

In most cases, the `direct_IO` method is a wrapper for the `__blockdev_direct_IO()` function. This function is quite complex and invokes a large number of auxiliary data structures and functions; however, it executes essentially the same kind of operations already described in this chapter: it splits the data to be read or written in suitable blocks, locates the data on disk, and fills up one or more bio descriptors that describe the I/O operations to be performed. Of course, the data will be read or written directly in the User Mode buffers specified by the `iovec` descriptors in the `iov` array. The bio descriptors are submitted to the generic block layer by invoking the `submit_bio()` function (see the section "Submitting Buffer Heads to the Generic Block Layer" in Chapter 15). Usually, the `__blockdev_direct_IO()` function does not return until all direct I/O transfers have been completed; thus, once the `read()` or `write()` system call returns, the self-caching application can safely access the buffers containing the file data.

Asynchronous I/O

The POSIX 1003.1 standard defines a set of library functions—listed in Table 16-4—for accessing the files in an asynchronous way. "Asynchronous" essentially means that when a User Mode process invokes a library function to read or write a file, the function terminates as soon as the read or write operation has been enqueued, possibly even before the actual I/O data transfer takes place. The calling process can thus continue its execution while the data is being transferred.

Table 16-4. The POSIX library functions for asynchronous I/O

Function	Description
`aio_read()`	Asynchronously reads some data from a file
`aio_write()`	Asynchronously writes some data into a file
`aio_fsync()`	Requests a flush operation for all outstanding asynchronous I/O operations (does not block)

Table 16-4. The POSIX library functions for asynchronous I/O (continued)

Function	Description
`aio_error()`	Gets the error code for an outstanding asynchronous I/O operation
`aio_return()`	Gets the return code for a completed asynchronous I/O operation
`aio_cancel()`	Cancels an outstanding asynchronous I/O operation
`aio_suspend()`	Suspends the process until at least one of several outstanding I/O operations completes

Using asynchronous I/O is quite simple. The application opens the file by means of the usual `open()` system call. Then, it fills up a control block of type `struct aiocb` with the information describing the requested operation. The most commonly used fields of the `struct aiocb` control block are:

`aio_fildes`
> The file descriptor of the file (as returned by the `open()` system call)

`aio_buf`
> The User Mode buffer for the file's data

`aio_nbytes`
> How many bytes should be transferred

`aio_offset`
> Position in the file where the read or write operation will start (it is independent of the "synchronous" file pointer)

Finally, the application passes the address of the control block to either `aio_read()` or `aio_write()`; both functions terminate as soon as the requested I/O data transfer has been enqueued by the system library or kernel. The application can later check the status of the outstanding I/O operation by invoking `aio_error()`, which returns `EINPROGRESS` if the data transfer is still in progress, 0 if it is successfully completed, or an error code in case of failure. The `aio_return()` function returns the number of bytes effectively read or written by a completed asynchronous I/O operation, or -1 in case of failure.

Asynchronous I/O in Linux 2.6

Asynchronous I/O can be implemented by a system library without any kernel support at all. Essentially, the `aio_read()` or `aio_write()` library function clones the current process and lets the child invoke the synchronous `read()` or `write()` system calls; then, the parent terminates the `aio_read()` or `aio_write()` function and continues the execution of the program, hence it does not wait for the synchronous operation started by the child to finish. However, this "poor man's" version of the POSIX functions is significantly slower than a version that uses a kernel-level implementation of asynchronous I/O.

The Linux 2.6 kernel version supports a set of system calls for asynchronous I/O. However, in Linux 2.6.11 this feature is a work in progress, and asynchronous I/O works properly only for files opened with the O_DIRECT flag set (see the previous section). The system calls for asynchronous I/O are listed in Table 16-5.

Table 16-5. Linux system calls for asynchronous I/O

System call	Description
io_setup()	Initializes an asynchronous context for the current process
io_submit()	Submits one or more asynchronous I/O operations
io_getevents()	Gets the completion status of some outstanding asynchronous I/O operations
io_cancel()	Cancels an outstanding I/O operation
io_destroy()	Removes an asynchronous context for the current process

The asynchronous I/O context

If a User Mode process wants to make use of the io_submit() system call to start an asynchronous I/O operation, it must create beforehand an *asynchronous I/O context*.

Basically, an asynchronous I/O context (in short, AIO context) is a set of data structures that keep track of the on-going progresses of the asynchronous I/O operations requested by the process. Each AIO context is associated with a kioctx object, which stores all information relevant for the context. An application might create several AIO contexts; all kioctx descriptors of a given process are collected in a singly linked list rooted at the ioctx_list field of the memory descriptor (see Table 9-2 in Chapter 9).

We are not going to discuss in detail the kioctx object; however, we should pinpoint an important data structure referenced by the kioctx object: the AIO ring.

The *AIO ring* is a memory buffer in the address space of the User Mode process that is also accessible by all processes in Kernel Mode. The User Mode starting address and length of the AIO ring are stored in the ring_info.mmap_base and ring_info.mmap_size fields of the kioctx object, respectively. The descriptors of all page frames composing the AIO ring are stored in an array pointed to by the ring_info.ring_pages field.

The AIO ring is essentially a circular buffer where the kernel writes the completion reports of the outstanding asynchronous I/O operations. The first bytes of the AIO ring contain an header (a struct aio_ring data structure); the remaining bytes store io_event data structures, each of which describes a completed asynchronous I/O operation. Because the pages of the AIO ring are mapped in the User Mode address space of the process, the application can check directly the progress of the outstanding asynchronous I/O operations, thus avoiding using a relatively slow system call.

The io_setup() system call creates a new AIO context for the calling process. It expects two parameters: the maximum number of outstanding asynchronous I/O

operations, which ultimately determines the size of the AIO ring, and a pointer to a variable that will store a handle to the context; this handle is also the base address of the AIO ring. The sys_io_setup() service routine essentially invokes do_mmap() to allocate a new anonymous memory region for the process that will contain the AIO ring (see the section "Allocating a Linear Address Interval" in Chapter 9), and creates and initializes a kioctx object describing the AIO context.

Conversely, the io_destroy() system call removes an AIO context; it also destroys the anonymous memory region containing the corresponding AIO ring. The system call blocks the current process until all outstanding asynchronous I/O operations are complete.

Submitting the asynchronous I/O operations

To start some asynchronous I/O operations, the application invokes the io_submit() system call. The system call has three parameters:

ctx_id
> The handle returned by io_setup(), which identifies the AIO context

iocbpp
> The address of an array of pointers to descriptors of type iocb, each of which describes one asynchronous I/O operation

nr
> The length of the array pointed to by iocbpp

The iocb data structure includes the same fields as the POSIX aiocb descriptor (aio_fildes, aio_buf, aio_nbytes, aio_offset) plus the aio_lio_opcode field that stores the type of the requested operation (typically read, write, or sync).

The service routine sys_io_submit() performs essentially the following steps:

1. Verifies that the array of iocb descriptors is valid.

2. Searches the kioctx object corresponding to the ctx_id handle in the list rooted at the ioctx_list field of the memory descriptor.

3. For each iocb descriptor in the array, it executes the following substeps:

    a. Gets the address of the file object corresponding to the file descriptor stored in the aio_fildes field.

    b. Allocates and initializes a new kiocb descriptor for the I/O operation.

    c. Checks that there is a free slot in the AIO ring to store the completion result of the operation.

    d. Sets the ki_retry method of the kiocb descriptor according to the type of the operation (see below).

    e. Executes the aio_run_iocb() function, which essentially invokes the ki_retry method to start the I/O data transfer for the corresponding asynchronous I/O operation. If the ki_retry method returns the value -EIOCBRETRY,

the asynchronous I/O operation has been submitted but not yet fully satisfied: the `aio_run_iocb()` function will be invoked again on this `kiocb` at a later time (see below). Otherwise, it invokes `aio_complete()` to add a completion event for the asynchronous I/O operation in the ring of the AIO context.

If the asynchronous I/O operation is a read request, the `ki_retry` method of the corresponding `kiocb` descriptor is implemented by `aio_pread()`. This function essentially executes the `aio_read` method of the file object, then updates the `ki_buf` and `ki_left` fields of the `kiocb` descriptor (see Table 16-1 earlier in this chapter) according to the value returned by the `aio_read` method. Finally, `aio_pread()` returns the number of bytes effectively read from the file, or the value `-EIOCBRETRY` if the function determines that not all requested bytes have been transferred. For most filesystems, the `aio_read` method of the file object ends up invoking the `__generic_file_aio_read()` function. Assuming that the `O_DIRECT` flag of the file is set, this function ends up invoking the `generic_file_direct_IO()` function, as described in the previous section. In this case, however, the `__blockdev_direct_IO()` function does not block the current process waiting for the I/O data transfer to complete; instead, the function returns immediately. Because the asynchronous I/O operation is still outstanding, the `aio_run_iocb()` will be invoked again, this time by the *aio* kernel thread of the `aio_wq` work queue. The `kiocb` descriptor keeps track of the progress of the I/O data transfer; eventually all requested data will be transferred and the completion result will be added to the AIO ring.

Similarly, if the asynchronous I/O operation is a write request, the `ki_retry` method of the `kiocb` descriptor is implemented by `aio_pwrite()`. This function essentially executes the `aio_write` method of the file object, then updates the `ki_buf` and `ki_left` fields of the `kiocb` descriptor (see Table 16-1 earlier in this chapter) according to the value returned by the `aio_write` method. Finally, `aio_pwrite()` returns the number of bytes effectively written to the file, or the value `-EIOCBRETRY` if the function determines that not all requested bytes have been transferred. For most filesystems, the `aio_write` method of the file object ends up invoking the `generic_file_aio_write_nolock()` function. Assuming that the `O_DIRECT` flag of the file is set, this function ends up invoking the `generic_file_direct_IO()` function, as above.

CHAPTER 17

Page Frame Reclaiming

In previous chapters, we explained how the kernel handles dynamic memory by keeping track of free and busy page frames. We have also discussed how every process in User Mode has its own address space and has its requests for memory satisfied by the kernel one page at a time, so that page frames can be assigned to the process at the very last possible moment. Last but not least, we have shown how the kernel makes use of dynamic memory to implement both memory and disk caches.

In this chapter, we complete our description of the virtual memory subsystem by discussing page frame reclaiming. We'll start in the first section, "The Page Frame Reclaiming Algorithm," explaining why the kernel needs to reclaim page frames and what strategy it uses to achieve this. We then make a technical digression in the section "Reverse Mapping" to discuss the data structures used by the kernel to locate quickly all the Page Table entries that point to the same page frame. The section "Implementing the PFRA" is devoted to the page frame reclaiming algorithm used by Linux. The last main section, "Swapping," is almost a chapter by itself: it covers the swap subsystem, a kernel component used to save anonymous (not mapping data of files) pages on disk.

The Page Frame Reclaiming Algorithm

One of the fascinating aspects of Linux is that the checks performed before allocating dynamic memory to User Mode processes or to the kernel are somewhat perfunctory.

No rigorous check is made, for instance, on the total amount of RAM assigned to the processes created by a single user (the limits mentioned in the section "Process Resource Limits" in Chapter 3 mostly affect single processes). Similarly, no limit is placed on the size of the many disk caches and memory caches used by the kernel.

This lack of controls is a design choice that allows the kernel to use the available RAM in the best possible way. When the system load is low, the RAM is filled mostly by the disk caches and the few running processes can benefit from the information

stored in them. However, when the system load increases, the RAM is filled mostly by pages of the processes and the caches are shrunken to make room for additional processes.

As we saw in previous chapters, both memory and disk caches grab more and more page frames but never release any of them. This is reasonable because cache systems don't know if and when processes will reuse some of the cached data and are therefore unable to identify the portions of cache that should be released. Moreover, thanks to the demand paging mechanism described in Chapter 9, User Mode processes get page frames as long as they proceed with their execution; however, demand paging has no way to force processes to release the page frames whenever they are no longer used.

Thus, sooner or later all the free memory will be assigned to processes and caches. The *page frame reclaiming* algorithm of the Linux kernel refills the lists of free blocks of the buddy system by "stealing" page frames from both User Mode processes and kernel caches.

Actually, page frame reclaiming must be performed *before* all the free memory has been used up. Otherwise, the kernel might be easily trapped in a deadly chain of memory requests that leads to a system crash. Essentially, to free a page frame the kernel must write its data to disk; however, to accomplish this operation, the kernel requires another page frame (for instance, to allocate the buffer heads for the I/O data transfer). If no free page frame exists, no page frame can be freed.

One of the goals of page frame reclaiming is thus to conserve a minimal pool of free page frames so that the kernel may safely recover from "low on memory" conditions.

Selecting a Target Page

The objective of the page frame reclaiming algorithm (*PFRA*) is to pick up page frames and make them free. Clearly the page frames selected by the PFRA must be *non-free*, that is, they must not be already included in one of the `free_area` arrays used by the buddy system (see the section "The Buddy System Algorithm" in Chapter 8).

The PFRA handles the page frames in different ways, according to their contents. We can distinguish between *unreclaimable pages*, *swappable pages*, *syncable pages*, and *discardable pages*. These types are explained in Table 17-1.

Table 17-1. The types of pages considered by the PFRA

Type of pages	Description	Reclaim action
Unreclaimable	Free pages (included in buddy system lists) Reserved pages (with `PG_reserved` flag set) Pages dynamically allocated by the kernel Pages in the Kernel Mode stacks of the processes Temporarily locked pages (with `PG_locked` flag set) Memory locked pages (in memory regions with `VM_LOCKED` flag set)	(No reclaiming allowed or needed)

Table 17-1. The types of pages considered by the PFRA (continued)

Type of pages	Description	Reclaim action
Swappable	Anonymous pages in User Mode address spaces Mapped pages of *tmpfs* filesystem (e.g., pages of IPC shared memory)	Save the page contents in a swap area
Syncable	Mapped pages in User Mode address spaces Pages included in the page cache and containing data of disk files Block device buffer pages Pages of some disk caches (e.g., the inode cache)	Synchronize the page with its image on disk, if necessary
Discardable	Unused pages included in memory caches (e.g., slab allocator caches) Unused pages of the dentry cache	Nothing to be done

In the above table, a page is said to be *mapped* if it maps a portion of a file. For instance, all pages in the User Mode address spaces belonging to file memory mappings are mapped, as well as any other page included in the page cache. In almost all cases, mapped pages are syncable: in order to reclaim the page frame, the kernel must check whether the page is dirty and, if necessary, write the page contents in the corresponding disk file.

Conversely, a page is said to be *anonymous* if it belongs to an anonymous memory region of a process (for instance, all pages in the User Mode heap or stack of a process are anonymous). In order to reclaim the page frame, the kernel must save the page contents in a dedicated disk partition or disk file called "swap area" (see the later section "Swapping"); therefore, all anonymous pages are swappable.

Usually, the pages of special filesystems are not reclaimable. The only exceptions are the pages of the *tmpfs* special filesystem, which can be reclaimed by saving them in a swap area. As we'll see in Chapter 19, the *tmpfs* special filesystem is used by the IPC shared memory mechanism.

When the PFRA must reclaim a page frame belonging to the User Mode address space of a process, it must take into consideration whether the page frame is *shared* or *non-shared*. A shared page frame belongs to multiple User Mode address spaces, while a non-shared page frame belongs to just one. Notice that a non-shared page frame might belong to several lightweight processes referring to the same memory descriptor.

Shared page frames are typically created when a process spawns a child; as explained in the section "Copy On Write" in Chapter 9, the page tables of the child are copied from those of the parent, thus parent and child share the same page frames. Another common case occurs when two or more processes access the same file by means of a shared memory mapping (see the section "Memory Mapping" in Chapter 16).[*]

---

[*] It should be noted, however, that when a single process accesses a file through a shared memory mapping, the corresponding pages are non-shared as far as the PFRA is concerned. Similarly, a page belonging to a private memory mapping may be treated as shared by the PFRA (for instance, because two processes read the same file portion and none of them modified the data in the page).

Design of the PFRA

While it is easy to identify the page candidates for memory reclaiming—roughly speaking, any page belonging to a disk or memory cache, or to the User Mode address space of a process—selecting the proper target pages is perhaps the most sensitive issue in kernel design.

As a matter of fact, the hardest job of a developer working on the virtual memory subsystem consists of finding an algorithm that ensures acceptable performance both for desktop machines (on which memory requests are quite limited but system responsiveness is crucial) and for high-level machines such as large database servers (on which memory requests tend to be huge).

Unfortunately, finding a good page frame reclaiming algorithm is a rather empirical job, with very little support from theory. The situation is somewhat similar to evaluating the factors that determine the dynamic priority of a process: the main objective is to tune the parameters in such a way to achieve good system performance, without asking too many questions about why it works well. Often, it's just a matter of "let's try this approach and see what happens." An unpleasant side effect of this empirical design is that the code changes quickly. For that reason, we cannot ensure that the memory reclaiming algorithm we are going to describe—the one used in Linux 2.6.11—will be exactly the same, by the time you'll read this chapter, as the one adopted by the most up-to-date version of the Linux 2.6 kernel. However, the general ideas and the main heuristic rules described here should continue to hold.

Looking too close to the trees' leaves might lead us to miss the whole forest. Therefore, let us present a few general rules adopted by the PFRA. These rules are embedded in the functions that will be described later in this chapter.

Free the "harmless" pages first
> Pages included in disk and memory caches not referenced by any process should be reclaimed before pages belonging to the User Mode address spaces of the processes; in the former case, in fact, the page frame reclaiming can be done without modifying any Page Table entry. As we will see in the section "The Least Recently Used (LRU) Lists" later in this chapter, this rule is somewhat mitigated by introducing a "swap tendency factor."

Make all pages of a User Mode process reclaimable
> With the exception of locked pages, the PFRA must be able to steal any page of a User Mode process, including the anonymous pages. In this way, processes that have been sleeping for a long period of time will progressively lose all their page frames.

Reclaim a shared page frame by unmapping at once all page table entries that reference it
> When the PFRA wants to free a page frame shared by several processes, it clears all page table entries that refer to the shared page frame, and then reclaims the page frame.

Reclaim "unused" pages only

The PFRA uses a simplified *Least Recently Used (LRU) replacement algorithm* to classify pages as *in-use* and *unused*.[*] If a page has not been accessed for a long time, the probability that it will be accessed in the near future is low and it can be considered "unused;" on the other hand, if a page has been accessed recently, the probability that it will continue to be accessed is high and it must be considered as "in-use." The PFRA reclaims only unused pages. This is just another application of the locality principle mentioned in the section "Hardware Cache" in Chapter 2.

The main idea behind the LRU algorithm is to associate a counter storing the age of the page with each page in RAM—that is, the interval of time elapsed since the last access to the page. This counter allows the PFRA to reclaim only the oldest page of any process. Some computer platforms provide sophisticated support for LRU algorithms;[†] unfortunately, 80×86 processors do not offer such a hardware feature, thus the Linux kernel cannot rely on a page counter that keeps track of the age of every page. To cope with this restriction, Linux takes advantage of the `Accessed` bit included in each Page Table entry, which is automatically set by the hardware when the page is accessed; moreover, the age of a page is represented by the position of the page descriptor in one of two different lists (see the section "The Least Recently Used (LRU) Lists" later in this chapter).

Therefore, the page frame reclaiming algorithm is a blend of several heuristics:

- Careful selection of the order in which caches are examined.
- Ordering of pages based on aging (least recently used pages should be freed before pages accessed recently).
- Distinction of pages based on the page state (for example, non-dirty pages are better candidates than dirty pages because they don't have to be written to disk).

Reverse Mapping

As stated in the previous section, one of the objectives of the PFRA is to be able to free a shared page frame. To that end, the Linux 2.6 kernel is able to locate quickly all the Page Table entries that point to the same page frame. This activity is called *reverse mapping*.

A trivial solution for reverse mapping would be to include in each page descriptor additional fields to link together all the Page Table entries that point to the page

---

[*] The PFRA could also be considered as a "used-once" algorithm, which has its roots in the 2Q buffer management replacement algorithm proposed by T. Johnson and D. Shasha in 1994.

[†] For instance, the CPUs of some mainframes automatically update the value of a counter included in each page table entry to specify the age of the corresponding page.

frame associated with the page descriptor. However, keeping such lists up-to-date would increase significantly the kernel overhead; for that reason, more sophisticated solutions have been devised. The technique used in Linux 2.6 is named *object-based reverse mapping*. Essentially, for any reclaimable User Mode page, the kernel stores the backward links to all memory regions in the system (the "objects") that include the page itself. Each memory region descriptor stores a pointer to a memory descriptor, which in turn includes a pointer to a Page Global Directory. Therefore, the backward links enable the PFRA to retrieve all Page Table entries referencing a given a page. Because there are fewer memory region descriptors than page descriptors, updating the backward links of a shared page is less time consuming. Let's see how this scheme is worked out.

First of all, the PFRA must have a way to determine whether the page to be reclaimed is shared or non-shared, and whether it is mapped or anonymous. In order to do this, the kernel looks at two fields of the page descriptor: _mapcount and mapping.

The _mapcount field stores the number of Page Table entries that refer to the page frame. The counter starts from -1: this value means that no Page Table entry references the page frame. Thus, if the counter is zero, the page is non-shared, while if it is greater than zero the page is shared. The page_mapcount() function receives the address of a page descriptor and returns the value of its _mapcount plus one (thus, for instance, it returns one for a non-shared page included in the User Mode address space of some process).

The mapping field of the page descriptor determines whether the page is mapped or anonymous, as follows:

- If the mapping field is NULL, the page belongs to the swap cache (see the section "The Swap Cache" later in this chapter).

- If the mapping field is not NULL and its least significant bit is 1, it means the page is anonymous and the mapping field encodes the pointer to an anon_vma descriptor (see the next section, "Reverse Mapping for Anonymous Pages").

- If the mapping field is non-NULL and its least significant bit is 0, the page is mapped; the mapping field points to the address_space object of the corresponding file (see the section "The address_space Object" in Chapter 15).

Every address_space object used by Linux is aligned in RAM so that its starting linear address is a multiple of four. Therefore, the least significant bit of the mapping field can be used as a flag denoting whether the field contains a pointer to an address_space object or to an anon_vma descriptor. This is a dirty programming trick, but the kernel uses a lot of page descriptors, thus these data structures should be as small as possible. The PageAnon() function receives as its parameter the address of a page descriptor and returns 1 if the least significant bit of the mapping field is set, 0 otherwise.

The try_to_unmap() function, which receives as its parameter a pointer to a page descriptor, tries to clear all the Page Table entries that point to the page frame

associated with that page descriptor. The function returns `SWAP_SUCCESS` (zero) if the function succeeded in removing any reference to the page frame from all Page Table entries, it returns `SWAP_AGAIN` (one) if some reference could not be removed, and returns `SWAP_FAIL` (two) in case of errors. The function is quite short:

```
int try_to_unmap(struct page *page)
{
    int ret;
    if (PageAnon(page))
        ret = try_to_unmap_anon(page);
    else
        ret = try_to_unmap_file(page);
    if (!page_mapped(page))
        ret = SWAP_SUCCESS;
    return ret;
}
```

The `try_to_unmap_anon()` and `try_to_unmap_file()` functions take care of anonymous pages and mapped pages, respectively. These functions will be described in the forthcoming sections.

Reverse Mapping for Anonymous Pages

Anonymous pages are often shared among several processes. The most common case occurs when forking a new process: as explained in the section "Copy On Write" in Chapter 9, all page frames owned by the parent—including the anonymous pages—are assigned also to the child. Another (quite unusual) case occurs when a process creates a memory region specifying both the `MAP_ANONYMOUS` and `MAP_SHARED` flag: the pages of such a region will be shared among the future descendants of the process.

The strategy to link together all the anonymous pages that refer to the same page frame is simple: the anonymous memory regions that include the page frame are collected in a doubly linked circular list. Be warned that, even if an anonymous memory region includes different pages, there always is just one reverse mapping list for all the page frames in the region.

When the kernel assigns the first page frame to an anonymous region, it creates a new `anon_vma` data structure, which includes just two fields: `lock`, a spin lock for protecting the list against race conditions, and `head`, the head of the doubly linked circular list of memory region descriptors. Then, the kernel inserts the `vm_area_struct` descriptor of the anonymous memory region in the `anon_vma`'s list; to that end, the `vm_area_struct` data structure includes two fields related to this list: `anon_vma_node` stores the pointers to the next and previous elements in the list, while `anon_vma` points to the `anon_vma` data structure. Finally, the kernel stores the address of the `anon_vma` data structure in the `mapping` field of the descriptor of the anonymous page, as described previously. See Figure 17-1.

When a page frame already referenced by one process is inserted into a Page Table entry of another process (for instance, as a consequence of a `fork()` system call, see
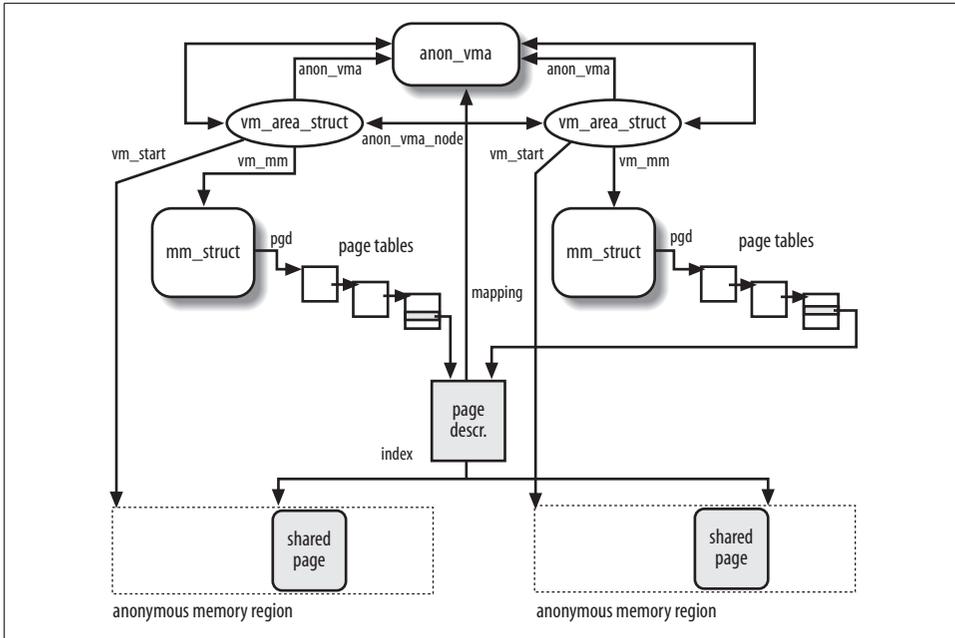
Figure 17-1. Object-based reverse mapping for anonymous pages

the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3); the kernel simply inserts the anonymous memory region of the second process in the doubly linked circular list of the anon_vma data structure pointed to by the anon_vma field of the first process's memory region. Therefore, any anon_vma's list typically includes memory regions owned by different processes.[*]

As shown in Figure 17-1, the anon_vma's list allows the kernel to quickly locate all Page Table entries that refer to the same anonymous page frame. In fact, each region descriptor stores in the vm_mm field the address of the memory descriptor, which in turn includes a field pgd containing the address of the Page Global Directory of the process. The Page Table entry can then be determined by considering the starting linear address of the anonymous page, which is easily obtained from the memory region descriptor and the index field of the page descriptor.

The try_to_unmap_anon( ) function

When reclaiming an anonymous page frame, the PFRA must scan all memory regions in the anon_vma's list and carefully check whether each region actually includes an anonymous page whose underlying page frame is the target page frame.

---

[*] An anon_vma's list may also include several adjacent anonymous memory regions owned by the same process. Usually this occurs when an anonymous memory region is split in two or more regions by the mprotect( ) system call.

This job is done by the try_to_unmap_anon() function, which receives as its parameter the descriptor of the target page frame and performs essentially the following steps:

1. Acquires the lock spin lock of the anon_vma data structure pointed to by the mapping field of the page descriptor.

2. Scans the anon_vma's list of memory region descriptors; for each vma memory region descriptor found in that list, it invokes the try_to_unmap_one() function passing as parameters vma and the page descriptor (see below). If for some reason this function returns a SWAP_FAIL value, or if the _mapcount field of the page descriptor indicates that all Page Table entries referencing the page frame have been found, the scanning terminates before reaching the end of the list.

3. Releases the spin lock obtained in step 1.

4. Returns the value computed by the last invocation of try_to_unmap_one(): SWAP_AGAIN (partial success) or SWAP_FAIL (failure).

The try_to_unmap_one() function

The try_to_unmap_one() function is called repeatedly both from try_to_unmap_anon() and from try_to_unmap_file(). It acts on two parameters: a pointer page to a target page descriptor and a pointer vma to a memory region descriptor. The function essentially performs the following actions:

1. Computes the linear address of the page to be reclaimed from the starting linear address of the memory region (vma->vm_start), the offset of the memory region in the mapped file (vma->vm_pgoff), and the offset of the page inside the mapped file (page->index). For anonymous pages, the vma->vm_pgoff field is either zero or equal to vm_start/PAGE_SIZE; correspondingly, the page->index field is either the index of the page inside the region or the linear address of the page divided by PAGE_SIZE.

2. If the target page is anonymous, it checks whether its linear address falls inside the memory region; if not, it terminates by returning SWAP_AGAIN. (As explained when introducing reverse mapping for anonymous pages, the anon_vma's list may include memory regions that do not contain the target page.)

3. Gets the address of the memory descriptor from vma->vm_mm, and acquires the vma->vm_mm->page_table_lock spin lock that protects the page tables.

4. Invokes successively pgd_offset(), pud_offset(), pmd_offset(), and pte_offset_map() to get the address of the Page Table entry that corresponds to the linear address of the target page.

5. Performs a few checks to verify that the target page is effectively reclaimable. If any of the following checks fails, the function jumps to step 12 to terminate by returning a proper error number, either SWAP_AGAIN or SWAP_FAIL:

a. Checks that the Page Table entry points to the target page; if not, the function returns `SWAP_AGAIN`. This can happen in the following cases:

- The Page Table entry refers to a page frame assigned with COW, but the anonymous memory region identified by `vma` still belongs to the `anon_vma` list of the original page frame.

- The `mremap()` system call may remap memory regions and move the pages into the User Mode address space by directly modifying the page table entries. In this particular case, object-based reverse mapping does not work, because the `index` field of the page descriptor cannot be used to determine the actual linear address of the page.

- The file memory mapping is non-linear (see the section "Non-Linear Memory Mappings" in Chapter 16).

b. Checks that the memory region is not locked (`VM_LOCKED`) or reserved (`VM_RESERVED`); if one of these restrictions is in place, the function returns `SWAP_FAIL`.

c. Checks that the `Accessed` bit inside the Page Table entry is cleared; if not, the function clears the bit and returns `SWAP_FAIL`. If the `Accessed` bit is set, the page is considered in-use, thus it should not be reclaimed.

d. Checks whether the page belongs to the swap cache (see the section "The Swap Cache" later in this chapter) and it is currently being handled by `get_user_pages()` (see the section "Allocating a Linear Address Interval" in Chapter 9); in this case, to avoid a nasty race condition, the function returns `SWAP_FAIL`.

6. The page can be reclaimed: if the `Dirty` bit in the Page Table entry is set, sets the `PG_dirty` flag of the page.

7. Clears the Page Table entry and flushes the corresponding TLBs.

8. If the page is anonymous, the function inserts a swapped-out page identifier in the Page Table entry so that further accesses to this page will swap in the page (see the section "Swapping" later in this chapter). Moreover, it decreases the counter of anonymous pages stored in the `anon_rss` field of the memory descriptor.

9. Decreases the counter of page frames allocated to the process stored in the `rss` field of the memory descriptor.

10. Decreases the `_mapcount` field of the page descriptor, because a reference to this page frame in the User Mode Page Table entries has been deleted.

11. Decreases the usage counter of the page frame, which is stored in the `_count` field of the page descriptor. If the counter becomes negative, it removes the page descriptor from the active or inactive list (see the section "The Least Recently Used (LRU) Lists" later in this chapter), and invokes `free_hot_page()` to release the page frame (see the section "The Per-CPU Page Frame Cache" in Chapter 8).

12. Invokes `pte_unmap()` to release the temporary kernel mapping that could have been allocated by `pte_offset_map()` in step 4 (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8).

13. Releases the `vma->vm_mm->page_table_lock` spin lock acquired in step 3.

14. Returns the proper error code (`SWAP_AGAIN` in case of success).

Reverse Mapping for Mapped Pages

As with anonymous pages, object-based reverse mapping for mapped pages is based on a simple idea: it is always possible to retrieve the Page Table entries that refer to a given page frame by accessing the descriptors of the memory regions that include the corresponding mapped pages. Thus, the core of reverse mapping is a clever data structure that collects all memory region descriptors relative to a given page frame.

We have seen in the previous section that descriptors for anonymous memory regions are collected in doubly linked circular lists; retrieving all page table entries referencing a given page frame involves a linear scanning of the elements in the list. The number of shared anonymous page frames is never very large, hence this approach works well.

Contrary to anonymous pages, mapped pages are frequently shared, because many different processes may share the same pages of code. For instance, consider that nearly all processes in the system share the pages containing the code of the standard C library (see the section "Libraries" in Chapter 20). For this reason, Linux 2.6 relies on special search trees, called "priority search trees," to quickly locate all the memory regions that refer to the same page frame.

There is a priority search tree for every file; its root is stored in the `i_mmap` field of the `address_space` object embedded in the file's `inode` object. It is always possible to quickly retrieve the root of the search tree, because the `mapping` field in the descriptor of a mapped page points to the `address_space` object.

The priority search tree

The *priority search tree* (*PST*) used by Linux 2.6 is based on a data structure introduced by Edward McCreight in 1985 to represent a set of overlapping intervals. McCreight's tree is a hybrid of a heap and a balanced search tree, and it is used to perform queries on the set of intervals—e.g., "what intervals are contained in a given interval?" and "what intervals intersect a given interval?"—in an amount of time directly proportional to the height of the tree and the number of intervals in the answer.

Each interval in a PST corresponds to a node of the tree, and it is characterized by two indices: the *radix index*, which corresponds to the starting point of the interval, and the *heap index*, which corresponds to the final point. The PST is essentially a

search tree on the radix index, with the additional heap-like property that the heap index of a node is never smaller than the heap indices of its children.

The Linux priority search tree differs from McCreight's data structure in two important aspects: first, the Linux tree is not always kept balanced (the balancing algorithm is costly both in memory space and in execution time); second, the Linux tree is adapted so as to store memory regions instead of linear intervals.

Each memory region can be considered as an interval of file pages identified by the initial position in the file (the radix index) and the final position (the heap index). However, memory regions tend to start from the same pages (typically, from page index 0). Unfortunately, McCreight's original data structure cannot store intervals having the very same starting point. As a partial solution, each node of a PST carries an additional *size index*—other than the radix and heap indices—corresponding to the size of the memory region in pages minus one. The size index allows the search program to distinguish different memory regions that start at the same file position.

The size index, however, increases significantly the number of different nodes that may end up in a PST. In particular, if there are too many nodes having the same radix index but different heap indices, the PST could not contain all of them. To solve this problem, the PST may include *overflow subtrees* rooted at the leaves of the PST and containing nodes having a common radix tree.

Furthermore, different processes may own memory regions that map exactly the same portion of the same file (just consider the example of the standard C library mentioned above). In that case, all nodes corresponding to these memory regions have the same radix, heap, and size indices. When the kernel must insert in a PST a memory region having the same indices as the ones of a node already existing, it inserts the memory region descriptor in a doubly linked circular list rooted at the older PST node.

Figure 17-2 shows a simple example of priority search tree. In the left side of the figure, we show seven memory regions covering the first six pages of a file; each interval is labeled with the radix index, size index, and heap index. In the right side of the figure, we draw the corresponding PST. Notice that no child node has a heap index greater than the heap index of the parent. Also observe that the radix index of the left child of any node is never greater than the radix index of the right child; in case of tie between the radix indices, the ordering is given by the size index. Let us suppose that the PFRA must retrieve all memory regions that include the page at index five. The search algorithm starts at the root (0,5,5): because the corresponding interval includes the page, this is the first retrieved memory region. Then the algorithm visits the left child (0,4,4) of the root and compares the heap index (four) with the page index: because the heap index is smaller, the interval does not include the page; moreover, thanks to the heap-like property of the PST, none of the children of this node can include the page. Thus the algorithm directly jumps to the right child (2,3,5) of the root. The corresponding interval includes the page, hence it is

retrieved. Then the algorithm visits the children (1,2,3) and (2,0,2), but it discovers that neither of them include the page.
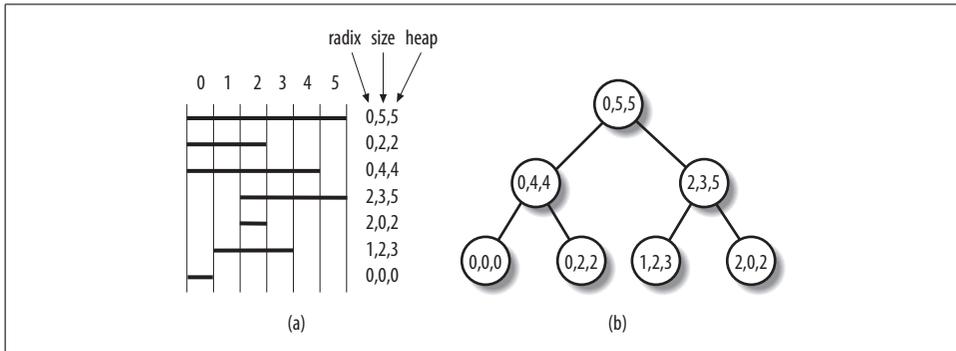
Figure 17-2. A simple example of priority search tree

We won't be able, for lack of space, to describe in detail the data structures and the functions that implement the Linux PSTs. We'll only mention that a node of a PST is represented by a prio_tree_node data structure, which is embedded in the shared. prio_tree_node field of each memory region descriptor. The shared.vm_set data structure is used—as an alternative to shared.prio_tree_node—to insert the memory region descriptor in a duplicate list of a PST node. PST nodes can be inserted and removed by executing the vma_prio_tree_insert() and vma_prio_tree_remove() functions; both of them receive as their parameters the address of a memory region descriptor and the address of a PST root. Queries on the PST can be performed by executing the vma_prio_tree_foreach macro, which implements a loop over all memory region descriptors that includes at least one page in a specified range of linear addresses.

The try_to_unmap_file( ) function

The try_to_unmap_file() function is invoked by try_to_unmap() to perform the reverse mapping of mapped pages. This function is quite simple to describe when the memory mapping is linear (see the section "Memory Mapping" in Chapter 16). In this case, it performs the following actions:

1. Gets the page->mapping->i_mmap_lock spin lock.
2. Applies the vma_prio_tree_foreach() macro to the priority search tree whose root is stored in the page->mapping->i_mmap field. For each vm_area_struct descriptor found by the macro, the function invokes try_to_unmap_one() to try to clear the Page Table entry of the memory region that contains the page (see the earlier section "Reverse Mapping for Anonymous Pages"). If for some reason this function returns a SWAP_FAIL value, or if the _mapcount field of the page descriptor indicates that all Page Table entries referencing the page frame have been found, the scanning terminates immediately.

3. Releases the `page->mapping->i_mmap_lock` spin lock.

4. Returns either `SWAP_AGAIN` or `SWAP_FAIL` according to whether all page table entries have been cleared.

If the mapping is non-linear (see the section "Non-Linear Memory Mappings" in Chapter 16), the `try_to_unmap_one()` function may fail to clear some Page Table entries, because the `index` field of the page descriptor, which as usual stores the position of the page in the file, is no longer related to the position of the page in the memory region. Therefore, `try_to_unmap_one()` cannot determine the linear address of the page, hence it cannot get the Page Table entry address.

The only solution is an exhaustive search in all the non-linear memory regions of the file. The doubly linked list rooted at the `i_mmap_nonlinear` field of the `page->mapping` file's `address_space` object includes the descriptors of all non-linear memory regions of the file. For each such memory region, `try_to_unmap_file()` invokes the `try_to_unmap_cluster()` function, which scans all Page Table entries corresponding to the linear addresses of the memory region and tries to clear them.

Because the search might be quite time-consuming, a limited scan is performed and a heuristic rule determines the portion of the memory region to be scanned: the `vm_private_data` field of the `vma_area_struct` descriptor holds the current cursor in the current scan. This means that `try_to_unmap_file()` might in some cases end up missing the page to be unmapped. When this occurs, `try_to_unmap()` discovers that the page is still mapped and return `SWAP_AGAIN` instead of `SWAP_SUCCESS`.

Implementing the PFRA

The page frame reclaiming algorithm must take care of many kinds of pages owned by User Mode processes, disk caches and memory caches; moreover, it has to obey several heuristic rules. Thus, it is not surprising that the PFRA is composed of a large number of functions. Figure 17-3 shows the main PFRA functions; an arrow denotes a function invocation, thus for instance `try_to_free_pages()` invokes `shrink_caches()`, `shrink_slab()`, and `out_of_memory()`.

As you can see, there are several "entry points" for the PFRA. Actually, page frame reclaiming is performed on essentially three occasions:

Low on memory reclaiming
    The kernel detects a "low on memory" condition.

Hibernation reclaiming
    The kernel must free memory because it is entering in the suspend-to-disk state (we don't further discuss this case).

Periodic reclaiming
    A kernel thread is activated periodically to perform memory reclaiming, if necessary.
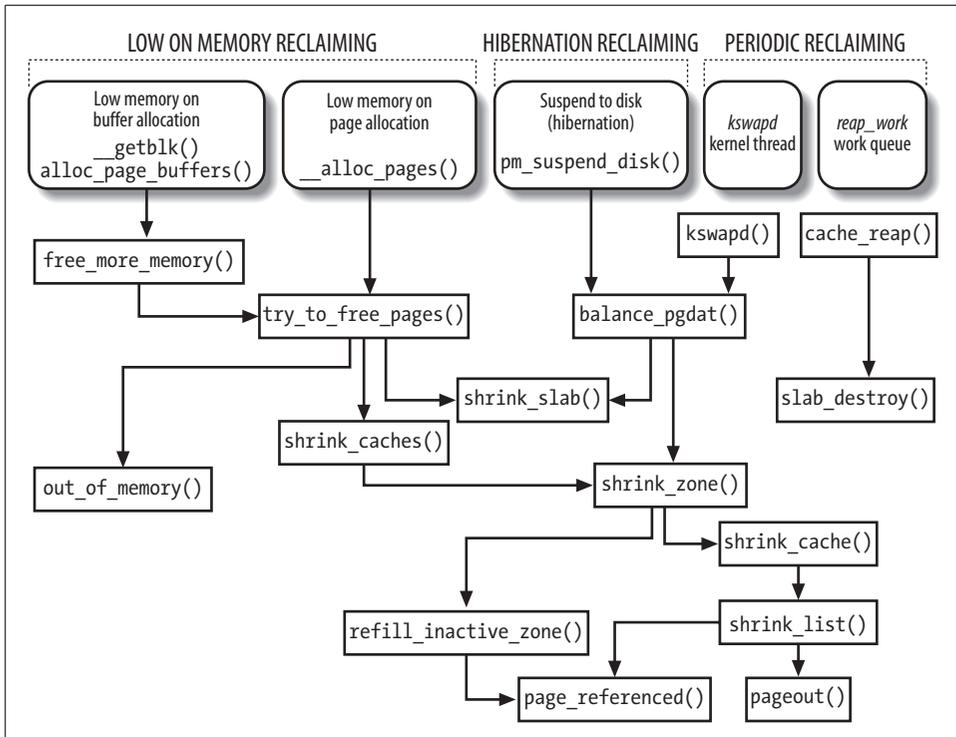
Figure 17-3. The main functions of the PFRA

Low on memory reclaiming is activated in the following cases:

- The grow_buffers() function, invoked by __getblk(), fails to allocate a new buffer page (see the section "Searching Blocks in the Page Cache" in Chapter 15).

- The alloc_page_buffers() function, invoked by create_empty_buffers(), fails to allocate the temporary buffer heads for a page (see the section "Reading and Writing a File" in Chapter 16).

- The __alloc_pages() function fails in allocating a group of contiguous page frames in a given list of memory zones (see the section "The Buddy System Algorithm" in Chapter 8).

Periodic reclaiming is activated by two different types of kernel threads:

- The *kswapd* kernel threads, which check whether the number of free page frames in some memory zone has fallen below the pages_high watermark (see the later section "Periodic Reclaiming").

- The *events* kernel threads, which are the worker threads of the predefined work queue (see the section "Work Queues" in Chapter 4); the PFRA periodically schedules the execution of a task in the predefined work queue to reclaim all free

slabs included in the memory caches handled by the slab allocator (see the section "The Slab Allocator" in Chapter 8).

We are now going to discuss in detail the various components of the page frame reclaiming algorithm, including all functions shown in Figure 17-3.

The Least Recently Used (LRU) Lists

All pages belonging to the User Mode address space of processes or to the page cache are grouped into two lists called the *active list* and the *inactive list*; they are also collectively denoted as *LRU lists*. The former list tends to include the pages that have been accessed recently, while the latter tends to include the pages that have not been accessed for some time. Clearly, pages should be stolen from the inactive list.

The active list and the inactive list of pages are the core data structures of the page frame reclaiming algorithm. The heads of these two doubly linked lists are stored, respectively, in the `active_list` and `inactive_list` fields of each zone descriptor (see the section "Memory Zones" in Chapter 8). The `nr_active` and `nr_inactive` fields in the same descriptor store the number of pages in the two lists. Finally, the `lru_lock` field is a spin lock that protects the two lists against concurrent accesses in SMP systems.

If a page belongs to an LRU list, its `PG_lru` flag in the page descriptor is set. Moreover, if the page belongs to the active list, the `PG_active` flag is set, while if it belongs to the inactive list, the `PG_active` flag is cleared. The `lru` field of the page descriptor stores the pointers to the next and previous elements in the LRU list.

Several auxiliary functions are available to handle the LRU lists:

`add_page_to_active_list()`
> Adds the page to the head of the zone's active list and increases the `nr_active` field of the zone descriptor.

`add_page_to_inactive_list()`
> Adds the page to the head of the zone's inactive list and increases the `nr_inactive` field of the zone descriptor.

`del_page_from_active_list()`
> Removes the page from the zone's active list and decreases the `nr_active` field of the zone descriptor.

`del_page_from_inactive_list()`
> Removes the page from the zone's inactive list and decreases the `nr_inactive` field of the zone descriptor.

`del_page_from_lru()`
> Checks the `PG_active` flag of a page; according to the result, removes the page from the active or inactive list, decreases the `nr_active` or `nr_inactive` field of the zone descriptor, and clears, if necessary, the `PG_active` flag.

`activate_page()`
> Checks the `PG_active` flag; if it is clear (the page is in the inactive list), it moves the page into the active list: invokes `del_page_from_inactive_list()`, then invokes `add_page_to_active_list()`, and finally sets the `PG_active` flag. The zone's `lru_lock` spin lock is acquired before moving the page.

`lru_cache_add()`
> If the page is not included in an LRU list, it sets the `PG_lru` flag, acquires the zone's `lru_lock` spin lock, and invokes `add_page_to_inactive_list()` to insert the page in the zone's inactive list.

`lru_cache_add_active()`
> If the page is not included in an LRU list, it sets the `PG_lru` and `PG_active` flags, acquires the zone's `lru_lock` spin lock, and invokes `add_page_to_active_list()` to insert the page in the zone's active list.

Actually, the last two functions, `lru_cache_add()` and `lru_cache_add_active()`, are slightly more complicated. In fact, the two functions do not immediately move the page into an LRU; instead, they accumulate the pages in temporary data structures of type `pagevec`, each of which may contain up to 14 page descriptor pointers. The pages will be effectively moved in an LRU list only when a `pagevec` structure is completely filled. This mechanism enhances the system performance, because the LRU spin lock is acquired only when the LRU lists are effectively modified.

Moving pages across the LRU lists

The PFRA collects the pages that were recently accessed in the active list so that it will not scan them when looking for a page frame to reclaim. Conversely, the PFRA collects the pages that have not been accessed for a long time in the inactive list. Of course, pages should move from the inactive list to the active list and back, according to whether they are being accessed.

Clearly, two page states ("active" and "inactive") are not sufficient to describe all possible access patterns. For instance, suppose a logger process writes some data in a page once every hour. Although the page is "inactive" for most of the time, the access makes it "active," thus denying the reclaiming of the corresponding page frame, even if it is not going to be accessed for an entire hour. Of course, there is no general solution to this problem, because the PFRA has no way to predict the behavior of User Mode processes; however, it seems reasonable that pages should not change their status on every single access.

The `PG_referenced` flag in the page descriptor is used to double the number of accesses required to move a page from the inactive list to the active list; it is also used to double the number of "missing accesses" required to move a page from the active list to the inactive list (see below). For instance, suppose that a page in the inactive list has the `PG_referenced` flag set to 0. The first page access sets the value of the flag to 1, but the page remains in the inactive list. The second page access finds the flag

set and causes the page to be moved in the active list. If, however, the second access does not occur within a given time interval after the first one, the page frame reclaiming algorithm may reset the PG_referenced flag.

As shown in Figure 17-4, the PFRA uses the mark_page_accessed(), page_referenced( ), and refill_inactive_zone() functions to move the pages across the LRU lists. In the figure, the LRU list including the page is specified by the status of the PG_active flag.
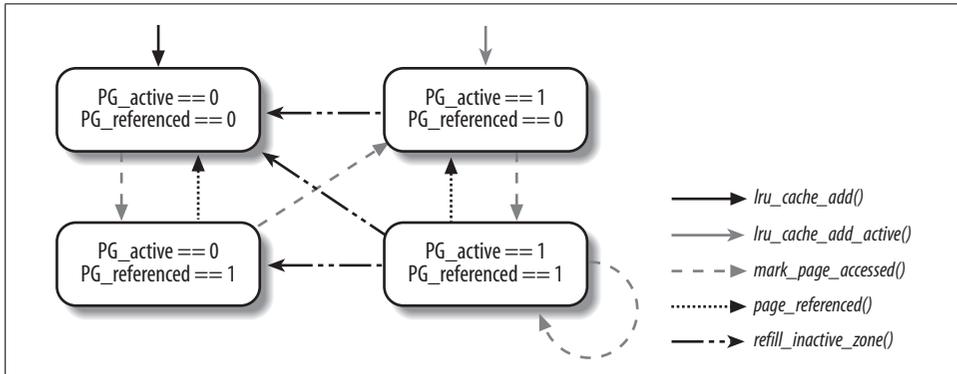
Figure 17-4. Moving pages across the LRU lists

The mark_page_accessed( ) function

Whenever the kernel must mark a page as accessed, it invokes the mark_page_ accessed( ) function. This happens every time the kernel determines that a page is being referenced by a User Mode process, a filesystem layer, or a device driver. For instance, mark_page_accessed( ) is invoked in the following cases:

- When loading on demand an anonymous page of a process (performed by the do_anonymous_page( ) function; see the section "Demand Paging" in Chapter 9).
- When loading on demand a page of a memory mapped file (performed by the filemap_nopage( ) function; see the section "Demand Paging for Memory Mapping" in Chapter 16).
- When loading on demand a page of an IPC shared memory region (performed by the shmem_nopage( ) function; see the section "IPC Shared Memory" in Chapter 19).
- When reading a page of data from a file (performed by the do_generic_file_ read( ) function; see the section "Reading from a File" in Chapter 16).
- When swapping in a page (performed by the do_swap_page( ) function; see the section "Swapping in Pages" later in this chapter).
- When looking up a buffer page in the page cache (see the __find_get_block( ) function in the section "Searching Blocks in the Page Cache" in Chapter 15).

The `mark_page_accessed()` function executes the following code fragment:

```
if (!PageActive(page) && PageReferenced(page) && PageLRU(page)) {
    activate_page(page);
    ClearPageReferenced(page);
} else if (!PageReferenced(page))
    SetPageReferenced(page);
```

As shown in Figure 17-4, the function moves the page from the inactive list to the active list only if the `PG_referenced` flag is set before the invocation.

The page_referenced( ) function

The `page_referenced()`function, which is invoked once for every page scanned by the PFRA, returns 1 if either the `PG_referenced` flag or some of the `Accessed` bits in the Page Table entries was set; it returns 0 otherwise. This function first checks the `PG_referenced` flag of the page descriptor; if the flag is set, it clears it. Next, it makes use of the object-based reverse mapping mechanism to check and clear the `Accessed` bits in all User Mode Page Table entries that refer to the page frame. To do this, the function makes use of three ancillary functions; `page_referenced_anon()`, `page_referenced_file()`, and `page_referenced_one()`, which are analogous to the `try_to_unmap_xxx()` functions described in the section "Reverse Mapping" earlier in this chapter. The `page_referenced()` function also honors the swap token; see the section "The Swap Token" later in this chapter.

The `page_referenced()` function never moves a page from the active list to the inactive list; this job is done by `refill_inactive_zone()`. In practice, this function does a lot more than move pages from the active to the inactive list, so we are going to describe it in greater detail.

The refill_inactive_zone( ) function

As illustrated in Figure 17-3, the `refill_inactive_zone()` function is invoked by `shrink_zone()`, which performs the reclaiming of pages in the page cache and in the User Mode address spaces (see the section "Low On Memory Reclaiming" later in this chapter). The function receives two parameters: a pointer `zone` to a memory zone descriptor, and a pointer `sc` to a `scan_control` structure. The latter data structure is widely used by the PFRA and contains information about the ongoing reclaiming operation; its fields are shown in Table 17-2.

Table 17-2. The fields of the scan_control descriptor

Type	Field	Description
unsigned long	nr_to_scan	Target number of pages to be scanned in the active list.
unsigned long	nr_scanned	Number of inactive pages scanned in the current iteration.
unsigned long	nr_reclaimed	Number of pages reclaimed in the current iteration.
unsigned long	nr_mapped	Number of pages referenced in the User Mode address spaces.

Table 17-2. The fields of the scan_control descriptor (continued)

Type	Field	Description
`int`	`nr_to_reclaim`	Target number of pages to be reclaimed.
`unsigned int`	`priority`	Priority of the scanning, ranging between 12 and 0. Lower priority implies scanning more pages.
`unsigned int`	`gfp_mask`	GFP mask passed from calling function.
`int`	`may_writepage`	If set, writing a dirty page to disk is allowed (only for laptop mode).

The role of `refill_inactive_zone()` is critical because moving a page from an active list to an inactive list means making the page eligible to fall prey, sooner or later, to the PFRA. If the function is too aggressive, it will move a lot of pages from the active list to the inactive list; as a consequence, the PFRA will reclaim a large number of page frames, and the system performance will be hit. On the other hand, if the function is too lazy, the inactive list will not be replenished with a large enough number of unused pages, and the PFRA will fail in reclaiming memory. Thus, the function implements an adaptive behavior: it starts by scanning, at every invocation, a small number of pages in the active list; however, if the PFRA is having trouble in reclaiming page frames, `refill_inactive_zone()` keeps increasing the number of active pages scanned at every invocation. This behavior is controlled by the value of the `priority` field in the `scan_control` data structure (a lower value means a more urgent priority).

Another heuristic rule regulates the behavior of the `refill_inactive_zone()` function. The LRU lists include two kinds of pages: those belonging to the User Mode address spaces, and those included in the page cache that do not belong to any User Mode process. As stated earlier, the PFRA should tend to shrink the page cache while leaving in RAM the pages owned by the User Mode processes. However, no fixed "golden rule" may yield good performance in every scenario, thus the `refill_inactive_zone()` function relies on a *swap tendency* heuristic value: it determines whether the function will move all kinds of pages, or just the pages that do not belong to the User Mode address spaces.[*] The swap tendency value is computed by the function as follows:

swap tendency = mapped ratio / 2 + distress + swappiness

The *mapped ratio* value is the percentage of pages in all memory zones that belong to User Mode address spaces (`sc->nr_mapped`) with respect to the total number of allocatable page frames. A high value of `mapped_ratio` means that the dynamic memory is mostly used by User Mode processes, while a low value means that it is mostly used by the page cache.

---

[*] The name "swap tendency" is a bit misleading, because the pages in User Mode address spaces can be swappable, syncable, or discardable. However, the swap tendency value certainly controls the amount of swapping performed by the PFRA, because almost all swappable pages belong to the User Mode address spaces.

The *distress* value is a measure of how effectively the PFRA is reclaiming page frames in this zone; it is based on the scanning priority of the zone in the previous run of the PFRA, which is stored in the `prev_priority` field of the zone descriptor. The distress value depends on the zone's previous priority as follows:

Zone prev. priority	12...7	6	5	4	3	2	1	0
Distress value	0	1	3	6	12	25	50	100

Finally, the *swappiness* value is a user-defined constant, which is usually set to 60. The system administrator may tune this value by writing in the */proc/sys/vm/ swappiness* file or by issuing the proper `sysctl()` system call.

Pages will be reclaimed from the address spaces of processes only if the zone's swap tendency is greater than or equal to 100. Thus, if the system administrator sets swappiness to 0, then the PFRA never reclaims pages in the User Mode address spaces unless the zone's previous priority is zero (an unlikely event); if the administrator sets swappiness to 100, then the PFRA reclaims pages in the User Mode address spaces at every invocation.

Here is a succinct description of what the `refill_inactive_zone()` function does:

1. Invokes `lru_add_drain()` to move into the active and inactive lists any page still contained in the `pagevec` data structures.

2. Gets the `zone->lru_lock` spin lock.

3. Performs a first cycle scanning the pages in `zone->active_list`, starting from the tail of the list and moving backwards. Continues until the list is empty or until `sc->nr_to_scan` pages have been scanned. For each page scanned in this cycle, the function increases by one its reference counter, removes the page descriptor from `zone->active_list`, and puts it in a temporary `l_hold` local list. However, if the reference counter of the page frame was zero, it puts back the page in the active list. In fact, page frames having a reference counter equal to zero should belong to the zone's Buddy system; however, to free a page frame, first its usage counter is decreased and then the page frame is removed from the LRU lists and inserted in the buddy system's list. Therefore, there is a small time window in which the PFRA may see a free page in an LRU list.

4. Adds to `zone->pages_scanned` the number of active pages that have been scanned.

5. Subtracts from `zone->nr_active` the number of pages that have been moved into the `l_hold` local list.

6. Releases the `zone->lru_lock` spin lock.

7. Computes the swap tendency value (see above).

8. Performs a second cycle on the pages in the `l_hold` local list. The objective of this cycle is to split the pages of `l_hold` into two local sublists called `l_active`

and `l_inactive`. A page belonging to the User Mode address space of some process—that is, a page whose `page->_mapcount` is nonnegative—is added to `l_active` if the swap tendency value is smaller than 100, or if the page is anonymous but no swap area is active, or finally if the `page_referenced()` function applied to the page returns a positive value, which means that the page has been recently accessed. In all other cases, the page is added to the `l_inactive` list.[*]

9. Gets the `zone->lru_lock` spin lock.

10. Performs a third cycle on the pages in the `l_inactive` local list to move them in the `zone->inactive_list` list and updates the `zone->nr_inactive` field. In doing so, it decreases the usage counters of the moved page frames to undo the increments done in step 3.

11. Performs a fourth and last cycle on the pages in the `l_active` local list to move them into the `zone->active_list` list and updates the `zone->nr_active` field. In doing so, it decreases the usage counters of the moved page frames to undo the increments done in step 3.

12. Releases the `zone->lru_lock` spin lock and returns.

It should be noted that `refill_inactive_zone()` checks the `PG_referenced` flag only for pages that belong to the User Mode address spaces (see step 8); in the opposite case, the pages are in the tail of the active list—hence they were accessed some time ago—and it is unlikely that they will be accessed in the near future. On the other hand, the function does not evict a page from the active list if it is owned by some User Mode process and has been recently used.

Low On Memory Reclaiming

Low on memory reclaiming is activated when a memory allocation fails. As shown in Figure 17-3, the kernel invokes `free_more_memory()` while allocating a VFS buffer or a buffer head, and it invokes `try_to_free_pages()` while allocating one or more page frames from the buddy system.

The free_more_memory() function

The `free_more_memory()` function performs the following actions:

1. Invokes `wakeup_bdflush()` to wake a *pdflush* kernel thread and trigger write operations for 1024 dirty pages in the page cache (see the section "The pdflush Kernel Threads" in Chapter 15). Writing dirty pages to disk may eventually make freeable the page frames containing buffers, buffers heads, and other VFS data structures.

---

[*] Notice that a page that does not belong to any User Mode process address space is moved into the inactive list; however, since its `PG_referenced` flag is not cleared, the first access to the page causes the `mark_page_accessed()` function to move the page back into the active list (see Figure 17-4).

2. Invokes the service routine of the `sched_yield()` system call to give the *pdflush* kernel thread a chance to run.

3. Starts a loop over all memory nodes in the system (see the section "Non-Uniform Memory Access (NUMA)" in Chapter 8). For each node, invokes the `try_to_free_pages()` function passing to it a list of the "low" memory zones (in the 80×86 architecture, `ZONE_DMA` and `ZONE_NORMAL`; see the section "Memory Zones" in Chapter 8).

The try_to_free_pages( ) function

The `try_to_free_pages()` function receives three parameters:

zones

A list of memory zones in which pages should be reclaimed (see the section "Memory Zones" in Chapter 8)

gfp_mask

The set of allocation flags that were used by the failed memory allocation (see the section "The Zoned Page Frame Allocator" in Chapter 8)

order

Not used

The goal of the function is to free at least 32 page frames by repeatedly invoking the `shrink_caches()` and `shrink_slab()` functions, each time with a higher priority than the previous invocation. The ancillary functions get the priority level—as well as other parameters of the ongoing scan operation—in a descriptor of type `scan_control` (see Table 17-2 earlier in this chapter). The lowest, initial priority level is 12, while the highest, final priority level is 0. If `try_to_free_pages()` does not succeed in reclaiming at least 32 page frames in one of the 13 repeated invocations of `shrink_caches()` and `shrink_slab()`, the PFRA is in serious trouble, and it has just one last resort: killing a process to free all its page frames. This operation is performed by the `out_of_memory()` function (see the section "The Out of Memory Killer" later in this chapter).

The function performs the following main steps:

1. Allocates and initializes a `scan_control` descriptor. In particular, stores the `gfp_mask` allocation mask in the `gfp_mask` field.

2. For each zone in the `zones` lists, it sets the `temp_priority` field of the zone descriptor to the initial priority (12). Moreover, it computes the total number of pages contained in the LRU lists of the zones.

3. Performs a loop of at most 13 iterations, from priority 12 down to 0; in each iteration performs the following substeps:

   a. Updates some field of the `scan_control` descriptor. In particular, it stores in the `nr_mapped` field the total number of pages owned by User Mode processes, and in the `priority` field the current priority of this iteration. Also, it sets to zero the `nr_scanned` and `nr_reclaimed` fields.

b. Invokes `shrink_caches()` passing as arguments the zones list and the address of the `scan_control` descriptor. This function scans the inactive pages of the zones (see below).

c. Invokes `shrink_slab()` to reclaim pages from the shrinkable kernel caches (see the section "Reclaiming Pages of Shrinkable Disk Caches" later in this chapter).

d. If the `current->reclaim_state` field is not `NULL`, it adds to the `nr_reclaimed` field of the `scan_control` descriptor the number of pages reclaimed from the slab allocator caches; this number is stored in a small data structure pointed to by the process descriptor field. The `__alloc_pages()` function sets up the `current->reclaim_state` field before invoking the `try_to_free_pages()` function, and clears the field right after its termination. (Oddly, the `free_more_memory()` function does not set this field.)

e. If the target has been reached (the `nr_reclaimed` field of the `scan_control` descriptor is greater than or equal to 32), it breaks the loop and jumps to step 4.

f. The target has not yet been reached. If at least 49 pages have been scanned so far, the function invokes `wakeup_bdflush()` to activate a *pdflush* kernel thread and write some dirty pages in the page cache to disk (see the section "Looking for Dirty Pages To Be Flushed" in Chapter 15).

g. If the function has already performed four iterations without reaching the target, it invokes `blk_congestion_wait()` to suspend the current process until any WRITE request queue becomes uncongested or until a 100 ms time-out elapses (see the section "Request Descriptors" in Chapter 14).

4. Sets the `prev_priority` field of each zone descriptor to the priority level used in the last invocation of `shrink_caches()`; it is stored in the `temp_priority` field of the zone descriptor.

5. Returns 1 if the reclaiming was successful, 0 otherwise.

The shrink_caches() function

The `shrink_caches()` function is invoked by `try_to_free_pages()`. It acts on two parameters: the zones list of memory zones, and the address `sc` of a `scan_control` descriptor.

The purpose of this function is simply to invoke the `shrink_zone()` function on each zone in the zones list. However, before invoking `shrink_zone()` on a given zone, `shrink_caches()` updates the `temp_priority` field of the zone's descriptor by using the value stored in the `sc->priority` field; this is the current priority level of the scanning operation. Moreover, if the priority value of the previous invocation of the PFRA is higher than the current priority value—that is, page frame reclaiming in this zone is now harder to do—`shrink_caches()` copies the current priority level into the `prev_priority` field of the zone descriptor. Finally, `shrink_caches()` does not invoke

shrink_zone() on a given zone if the all_unreclaimable flag in the zone descriptor is set and the current priority level is less than 12—that is, shrink_caches() is not being invoked in the very first iteration of try_to_free_pages(). The PFRA sets the all_unreclaimable flag when it decides that a zone is so full of unreclaimable pages that scanning the zone's pages is just a waste of time.

The shrink_zone() function

The shrink_zone() function acts on two parameters: zone, a pointer to a struct_zone descriptor, and sc, a pointer to a scan_control descriptor. The goal of this function is to reclaim 32 pages from the zone's inactive list; the function tries to reach this goal by invoking repeatedly an auxiliary function called shrink_cache(), each time on larger portion of the zone's inactive list. Moreover, shrink_zone() replenishes the zone's inactive list by repeatedly invoking the refill_inactive_zone() function described in the earlier section "The Least Recently Used (LRU) Lists."

The nr_scan_active and nr_scan_inactive fields of the zone descriptor play a special role here. To be efficient, the function works on batches of 32 pages. Thus, if the function is running at a low privilege level (high value of sc->priority) and one of the LRU lists does not contain enough pages, the function skips the scanning on that list. However, the number of active or inactive pages thus skipped is recorded in nr_scan_active or nr_scan_inactive, so that the skipped pages will be considered in the next invocation of the function.

Specifically, the shrink_zone() function performs the following steps:

1. Increases the zone->nr_scan_active by a fraction of the total number of elements in the active list (zone->nr_active). The actual increment is determined by the current priority level and ranges from zone->nr_active/2^{12} to zone->nr_active/2^0 (i.e., the whole number of active pages in the zone).

2. Increases the zone->nr_scan_inactive by a fraction of the total number of elements in the active list (zone->nr_inactive). The actual increment is determined by the current priority level and ranges from zone->nr_inactive/2^{12} to zone->nr_inactive.

3. If the zone->nr_scan_active field is greater than or equal to 32, the function copies its value in the nr_active local variable and sets the field to zero; otherwise, it sets nr_active to zero.

4. If the zone->nr_scan_inactive field is greater than or equal to 32, the function copies its value in the nr_inactive local variable and sets the field to zero; otherwise, it sets nr_inactive to zero.

5. Sets the sc->nr_to_reclaim field of the scan_control descriptor to 32.

6. If both nr_active and nr_inactive are 0, there is nothing to be done: the function terminates. This is an unlikely situation where User Mode processes have no page frames allocated to them.

7. If `nr_active` is positive, it replenishes the zone's inactive list:

```
sc->nr_to_scan = min(nr_active, 32);
nr_active -= sc->nr_to_scan;
refill_inactive_zone(zone, sc);
```

8. If `nr_inactive` is positive, it tries to reclaim at most 32 pages from the inactive list:

```
sc->nr_to_scan = min(nr_inactive, 32);
nr_inactive -= sc->nr_to_scan;
shrink_cache(zone, sc);
```

9. If `shrink_zone()` succeeds in reclaiming 32 pages (`sc->nr_to_reclaim` is now zero or negative), it terminates. Otherwise, it jumps back to step 6.

The shrink_cache( ) function

The `shrink_cache()` function is yet another auxiliary function whose main purpose is to extract from the zone's inactive list a group of pages, put them in a temporary list, and invoke the `shrink_list()` function to effectively perform page frame reclaiming on every page in that list. The `shrink_cache()` function acts on the same parameters as `shrink_zones()`, namely zone and sc, and performs the following main steps:

1. Invokes `lru_add_drain()` to move into the active and inactive lists any page still contained in the pagevec data structures (see the section "The Least Recently Used (LRU) Lists" earlier in this chapter).

2. Gets the `zone->lru_lock` spin lock.

3. Considers at most 32 pages in the inactive list; for each page, the function increases its usage counter, checks whether the page is not being freed to the buddy system (see the discussion at step 3 of `refill_inactive_zone()`), and moves the page from the zone's inactive list to a local list.

4. Decreases the counter `zone->nr_inactive` by the number of pages removed from the inactive list.

5. Increases the counter `zone->pages_scanned` by the number of pages effectively examined in the inactive list.

6. Releases the `zone->lru_lock` spin lock.

7. Invokes the `shrink_list()` function passing to it the (local list of) pages collected in step 3 above. This function is discussed below (as you were no doubt expecting).

8. Decreases the `sc->nr_to_reclaim` field by the number of pages actually reclaimed by `shrink_list()`.

9. Gets again the `zone->lru_lock` spin lock.

10. Puts back in the inactive or active list all pages of the local list that `shrink_list()` did not succeed in freeing. Notice that `shrink_list()` might mark a page as active by setting its `PG_active` flag. This operation is performed in a batch of pages using

a pagevec data structure (see the section "The Least Recently Used (LRU) Lists" earlier in this chapter).

11. If the function scanned at least `sc->nr_to_scan` pages, and if it didn't succeed in reclaiming the target number of pages (i.e., `sc->nr_to_reclaim` is still positive), it jumps back to step 3.

12. Releases the `zone->lru_lock` spin lock and terminates.

The shrink_list( ) function

We have now reached the heart of page frame reclaiming. While the purpose of the functions illustrated so far, from `try_to_free_pages()` to `shrink_cache()`, was to select the proper set of pages candidates for reclaiming, the `shrink_list()` function effectively tries to reclaim the pages passed as a parameter in the `page_list` list. The second parameter, namely `sc`, is the usual pointer to a `scan_control` descriptor. When `shrink_list()` returns, `page_list` contains the pages that couldn't be freed.

The function performs the following actions:

1. If the `need_resched` field of the current process is set, it invokes `schedule()`.

2. Starts a cycle on every page descriptor included in the `page_list` list. For each list item, it removes the page descriptor from the list and tries to reclaim the page frame; if for some reason the page frame could not be freed, it inserts the page descriptor in a local list.

3. Now the `page_list` list is empty: the function moves back the page descriptors from the local list to the `page_list` list.

4. Increases the `sc->nr_reclaimed` field by the number of page frames reclaimed in step 2, and returns that number.

Of course, what is really interesting in `shrink_list()` is the code that tries to reclaim a page frame. The flow diagram of this code is shown in Figure 17-5.

There are only three possible outcomes for each page frame handled by `shrink_list()`:

• The page is released to the zone's buddy system by invoking the `free_cold_page( )` function (see the section "The Per-CPU Page Frame Cache" in Chapter 8); hence, the page is effectively reclaimed.

• The page is not reclaimed, thus it will be reinserted in the `page_list` list; however, `shrink_list()` assumes that it will be possible to reclaim the page in the near future. Thus, the function leaves the `PG_active` flag in the page descriptor cleared, so that the page will be put back in the inactive list of the memory zone (see step 9 in the descriptor of `shrink_cache()` above). This event corresponds to the small boxes labeled as "INACTIVE" in Figure 17-5.

• The page is not reclaimed, thus it will be reinserted in the `page_list` list; however, either the page is in active use, or `shrink_list()` assumes that it will be impossible to reclaim the page in the foreseeable future. Thus, the function sets the `PG_active` flag in the page descriptor, so that the page will be put in the

active list of the memory zone. This event corresponds to the small boxes labeled as "ACTIVE" in Figure 17-5.

The shrink_list() function never tries to reclaim a page that is locked (PG_locked flag set) or under writeback (PG_writeback flag set). In order to test whether the page was recently referenced, shrink_list() invokes page_referenced(), which was described in the section "The Least Recently Used (LRU) Lists" earlier in this chapter.
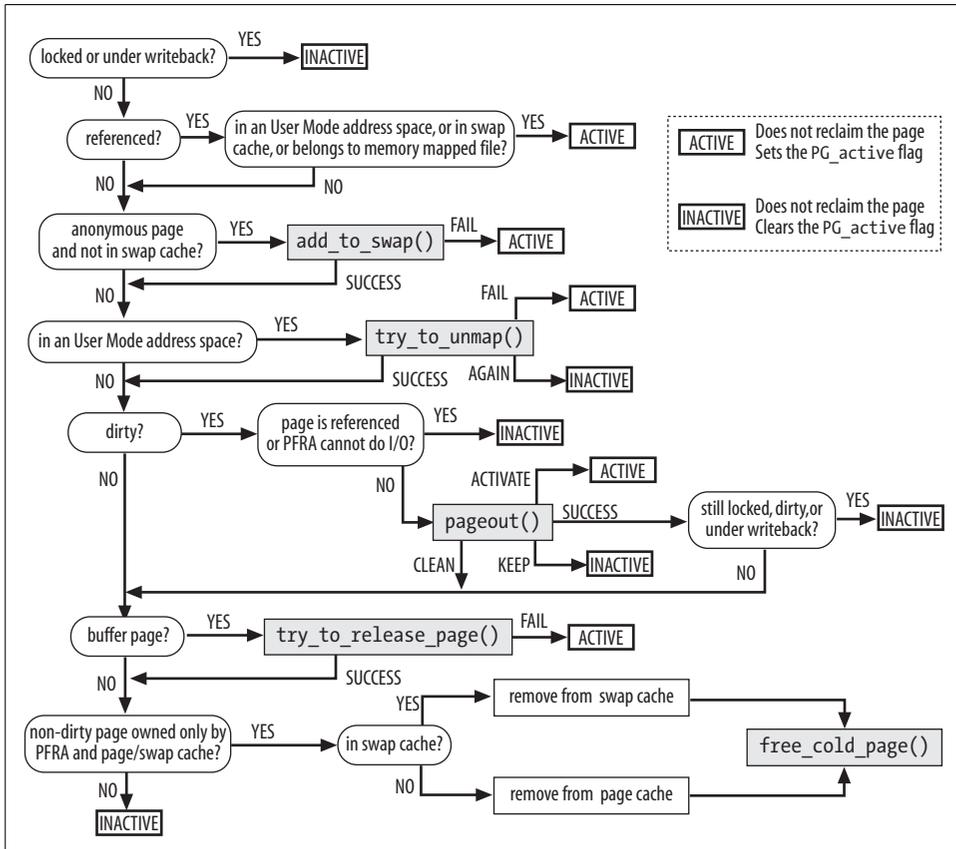
Figure 17-5. The page reclaiming logic of the shrink_list() function

To reclaim an anonymous page, the page must be added to the swap cache, and a new slot in a swap area must be reserved for it; see the section "Swapping" later in this chapter for details.

If the page is in the User Mode address space of some process (the _mapcount field in the page descriptor is greater than or equal to zero), shrink_list() invokes the try_to_unmap() function to locate all User Mode Page Table entries that refer to the page frame (see the section "Reverse Mapping" earlier in this chapter). Of course, reclaiming may proceed only if this function returns SWAP_SUCCESS.

If the page is dirty, it cannot be reclaimed unless it is written to disk. To do this, shrink_list() relies on the pageout() function, which is described next. The reclaiming of the page frame may proceed only if either pageout() does not have to issue a write operation, or if the write operation finishes soon.

If the page contains VFS buffers, shrink_list() invokes try_to_release_page() to release the associated buffer heads (see the section "Releasing Block Device Buffer Pages" in Chapter 15).

Finally, if everything went smoothly, shrink_list() checks the reference counter of the page: if it is equal to two, the page has just two owners: the page cache (or the swap cache, in case of anonymous pages), and the PFRA itself (the reference counter was increased in step 3 of shrink_cache(); see earlier). In this case, the page can be reclaimed, provided it is still not dirty. To do this, first the page is removed from the page cache or the swap cache, according to the value of the PG_swapcache flag of the page descriptor; then, the free_cold_page() function is executed.

The pageout() function

The pageout() function is invoked by shrink_list() when a dirty page must be written to disk. Essentially, the function performs the following operations:

1. Checks that the page is included in the page cache or in the swap cache (see the section "The Swap Cache" later in this chapter). Moreover, checks that the page is owned only by the page cache—or the swap cache—and the PFRA. Returns PAGE_KEEP if a check has failed (it does not make sense to write the page to disk if it is not reclaimable by shrink_list()).

2. Checks that the writepage method of the address_space object is defined; returns PAGE_ACTIVATE otherwise.

3. Checks that the current process can issue write requests to the request queue of the block device associated with the address_space object. Essentially, the *kswapd* and *pdflush* kernel threads may always issue the write request; normal processes can issue the write request only if the request queue is not congested, unless the current->backing_dev_info field points to the backing_dev_info data structure of the block device (see step 3 of the description of the generic_file_aio_write_nolock() function in the section "Writing to a File" in Chapter 16).

4. Checks that the page is still dirty; if not, returns PAGE_CLEAN.

5. Sets up a writeback_control descriptor and invokes the writepage method of the address_space object to start a write back operation (see the section "Writing Dirty Pages to Disk" in Chapter 16).

6. If the writepage method returned an error code, the function returns PAGE_ACTIVATE.

7. Returns PAGE_SUCCESS.

Reclaiming Pages of Shrinkable Disk Caches

We know from the previous chapters that the kernel uses other disk caches beside the page cache, for instance the dentry cache and the inode cache (see the section "The dentry Cache" in Chapter 12). When the PFRA tries to reclaim page frames, it should also check whether some of these disk caches can be shrunk.

Every disk cache that is considered by the PFRA must have a *shrinker function* registered at initialization time. The shrinker function expects two parameters: the target number of page frames to be reclaimed, and a set of GFP allocation flags; the function does what is required to reclaim the pages from the disk cache, then it returns the number of reclaimable pages remaining in the cache.

The set_shrinker() function registers a shrinker function with the PFRA. This function allocates a descriptor of type shrinker, stores the address of the shrinker function in the descriptor, and then inserts the descriptor in a global list rooted at the shrinker_list global variable. The set_shrinker() function also initializes the seeks field of the shrinker descriptor: informally, it is a parameter that indicates how much it costs to re-create one item of the cache once it is removed.

In Linux 2.6.11 there are few disk caches registered with the PFRA: besides the dentry cache and the inode cache, only the disk quota layer, the filesystem meta information block cache (mainly used for filesystems' extended attributes), and the XFS journaling filesystem register shrinker functions.

The PFRA's function that reclaims pages from the shrinkable disk caches is called shrink_slab() (the name is a bit misleading, because the function has little to do with the slab allocator caches). This function is invoked by try_to_free_pages(), as explained in the earlier section "Low On Memory Reclaiming," and by balance_pgdat(), which is described in the later section "Periodic Reclaiming."

The shrink_slab() function tries to balance the cost of reclaiming from the shrinkable disk cache with the cost of reclaiming from the LRU lists (performed by shrink_list()). Essentially, the function walks the list in the shrinker descriptors to invoke the shrinker functions and get the total number of reclaimable pages in the disk caches. Then, the function scans again the list of the shrinker descriptor; for each shrinkable disk cache, the function heuristically computes a target number of page frames to be reclaimed—based on the number of reclaimable pages in the disk caches, on the relative cost of re-creating a page in the disk cache, and on the number of pages in the LRU lists—and invokes the shrinker function to try to reclaim batches of at least 128 pages.

For lack of space, we'll limit ourselves to describe briefly the shrinker functions of the dentry cache and of the inode cache.

Reclaiming page frames from the dentry cache

The `shrink_dcache_memory()` function is the shrinker function for the dentry cache; it searches the cache for unused dentry objects—that is, objects not referenced by any process, see the section "dentry Objects" in Chapter 12—and releases them.

Because the dentry cache objects are allocated through the slab allocator, the `shrink_dcache_memory()` function may lead some slabs to become free, causing some page frames to be consequently reclaimed by `cache_reap()` (see the section "Periodic Reclaiming" later in this chapter). Moreover, the dentry cache acts as a controller of the inode cache. Therefore, when a dentry object is released, the pages storing the corresponding inode may become unused, and thus eventually released.

The `shrink_dcache_memory()` function receives as its parameters the number of page frames to reclaim and a GFP mask. It starts by checking whether the `__GFP_FS` bit in the GFP mask is clear; if so, the function returns -1, because releasing a dentry may trigger an operation on a disk-based filesystem. Page frame reclaiming is effectively done by invoking `prune_dcache()`. This function scans the list of unused dentries—whose head is stored in the `dentry_unused` variable—until it reaches the requested number of freed objects or until the whole list is scanned. On each object that wasn't recently referenced, the function:

1. Removes the dentry object from the dentry hash table, from the list of dentry objects in its parent directory, and from the list of dentry objects of the owner inode.

2. Decreases the usage counter of the dentry's inode by invoking the `d_iput` dentry method, if defined, or the `iput()` function.

3. Invokes the `d_release` method of the dentry object, if defined.

4. Invokes the `call_rcu()` function to register a callback function that will remove the dentry object (see the section "Read-Copy Update (RCU)" in Chapter 5). The callback function, in turn, will invoke `kmem_cache_free()` to release the object to the slab allocator (see the section "Freeing a Slab Object" in Chapter 8).

5. Decreases the usage counter of the parent directory.

Finally, `shrink_dcache_memory()` returns a value based on the number of unused dentries still contained in the dentry cache. More precisely, the returned value is the number of unused dentries multiplied by 100 and divided by the content of the `sysctl_vfs_cache_pressure` global variable. By default, this variable is equal to 100, thus the returned value is essentially the number of unused dentries. However, the system administrator may modify the variable by writing in the */proc/sys/vm/vfs_cache_pressure* or by issuing a suitable `sysctl()` system call. Setting this variable to a value smaller than 100 causes `shrink_slab()` to reclaim fewer pages from the dentry cache (and the inode cache; see the next section) with respect to the pages reclaimed from the LRU lists; conversely, setting the variable to a value greater than 100 causes

`shrink_slab()` to reclaim more pages from the dentry and inode caches with respect to the pages reclaimed from the LRU lists.

Reclaiming page frames from the inode cache

The `shrink_icache_memory()` function is invoked to remove unused inode objects from the inode cache; here, "unused" means that the inode no longer has a controlling dentry object. The function is similar to the `shrink_dcache_memory()` described previously. It checks the `__GFP_FS` bit in the `gfp_mask` parameter, then it invokes the `prune_icache()` function, and finally it returns a value based both on the number of unused inodes still included in the inode cache and the value of the `sysctl_vfs_cache_pressure` variable, as previously.

The `prune_icache()` function, in turn, scans the `inode_unused` list (see the section "Inode Objects" in Chapter 12); to free an inode, the function releases any private buffer associated with the inode, it invalidates the clean page frames in the page cache that refer to the inode and are not longer in use, and then it makes use of the `clear_inode()` and `destroy_inode()` functions to destroy the inode object.

Periodic Reclaiming

The PFRA performs periodic reclaiming by using two different mechanisms: the *kswapd* kernel threads, which invoke `shrink_zone()` and `shrink_slab()` to reclaim pages from the LRU lists, and the `cache_reap` function, which is invoked periodically to reclaim unused slabs from the slab allocator.

The kswapd kernel threads

The *kswapd* kernel threads are another kernel mechanism that activates page frame reclaiming. Why is it necessary? Is it not sufficient to invoke `try_to_free_pages()` when free memory becomes really scarce and another memory allocation request is issued?

Unfortunately, this is not the case. Some memory allocation requests are performed by interrupt and exception handlers, which cannot block the current process waiting for a page frame to be freed; moreover, some memory allocation requests are done by kernel control paths that have already acquired exclusive access to critical resources and that, therefore, cannot activate I/O data transfers. In the infrequent case in which all memory allocation requests are done by such sorts of kernel control paths, the kernel is never able to free memory.

The *kswapd* kernel threads also have a beneficial effect on system performance by keeping memory free in what would otherwise be idle time for the machine; processes can thus get their pages much faster.

There is a different *kswapd* kernel thread for each memory node (see the section "Non-Uniform Memory Access (NUMA)" in Chapter 8). Each such thread is usually

sleeping in the wait queue headed at the `kswapd_wait` field of the node descriptor. However, if `__alloc_pages()` discovers that all memory zones suitable for a memory allocation have a number of free page frames below a "warning" threshold—essentially, a value based on the `pages_low` and `protection` fields of the memory zone descriptor—then the function wakes up the *kswapd* kernel threads of the corresponding memory nodes (see the section "The Zone Allocator" in Chapter 8.) Essentially, the kernel starts to reclaim some page frames in order to avoid much more dramatic "low on memory" conditions.

As explained in the section "The Pool of Reserved Page Frames" in Chapter 8, every zone descriptor also includes a `pages_min` field—a threshold that specifies the minimum number of free page frames that should always be preserved—and a `pages_high` field—a threshold that specifies the "safe" number of free page frames above which page frame reclaiming should be stopped.

The *kswapd* kernel thread executes the `kswapd()` function. It initializes the kernel thread by binding the kernel thread to the CPUs that may access the memory node, by storing in the `current->reclaim_state` field of the process descriptor the address of a `reclaim_state` descriptor (see step 3d in the description of `try_to_free_pages()` earlier in this chapter), and by setting the `PF_MEMALLOC` and `PF_KSWAP` flags in the `current->flags` field—these flags indicate that the process is reclaiming memory and that it is allowed to use all the free memory available when doing its job. Every time the *kswapd* kernel thread is awakened, the `kswapd()` function performs essentially the following steps:

1. Invokes `finish_wait()` to remove the kernel thread from the node's `kswapd_wait` wait queue (see the section "How Processes Are Organized" in Chapter 3).

2. Invokes `balance_pgdat()` to perform the memory reclaiming on the *kswapd*'s memory node (see below).

3. Invokes `prepare_to_wait()` to set the process in the `TASK_INTERRUPTIBLE` state and to put it to sleep in the node's `kswapd_wait` wait queue.

4. Invokes `schedule()` to yield the CPU to some other runnable process.

The `balance_pgdat()` function performs, in turn, the following basic steps:

1. Sets up a `scan_control` descriptor (see Table 17-2 earlier in this chapter).

2. Sets the `temp_priority` field of each zone descriptor in the memory node to 12 (lowest priority).

3. Performs a loop of at most 13 iterations, from priority 12 down to 0; in each iteration performs the following substeps:

   a. Scans the memory zones to find the highest zone (from `ZONE_DMA` to `ZONE_HIGHMEM`) having an insufficient number of free page frames. Each test is done by executing the `zone_watermark_ok()` function described in the section "The Zone Allocator" in Chapter 8. If all zones have a large number of free page frames, it jumps to step 4.

b. Scans again the memory zones proceeding from `ZONE_DMA` to the zone found in step 3a. For each zone, it updates, if necessary, the `prev_priority` field of the zone descriptor with the current priority level, and invokes successively `shrink_zone()` to reclaim pages from the zone (see the earlier section "Low On Memory Reclaiming"). Next, it invokes `shrink_slab()` to reclaim pages from the shrinkable disk caches (see the earlier section "Reclaiming Pages of Shrinkable Disk Caches").

c. If at least 32 pages have been reclaimed, it breaks the loop and jumps to step 4.

4. Updates the `prev_priority` field of each zone descriptor with the value stored in the corresponding `temp_priority` field.

5. If some "low on memory" zone still exists, it invokes `schedule()` if the `need_resched` field of the process is set; when in execution again, it jumps back to step 1.

6. Returns the number of pages reclaimed.

The cache_reap( ) function

The PFRA must also reclaim the pages owned by the slab allocator caches (see the section "Memory Area Management" in Chapter 8). To do this, it relies on the `cache_reap()` function, which is periodically scheduled—approximately once every two seconds—in the predefined *events* work queue (see the section "Work Queues" in Chapter 4). The address of the `cache_reap()` function is stored in the `func` field of the `reap_work` per-CPU variable of type `work_struct`.

The `cache_reap()` function essentially performs the following steps:

1. Tries to acquire the `cache_chain_sem` semaphore, which protects the list of slab cache descriptors; if the semaphore is already taken, it invokes `schedule_delayed_work()` to schedule the next invocation of the function, and terminates.

2. Otherwise, scans the `kmem_cache_t` descriptors collected in the `cache_chain` list. For each cache descriptor found, the function performs the following steps:

    a. If the `SLAB_NO_REAP` flag in the cache descriptor is set, page frame reclaiming has been disabled, hence it continues with the next cache in the list.

    b. Drains the slab local cache (see the section "Local Caches of Free Slab Objects" in Chapter 8); this operation could cause new slabs to become free.

    c. Each cache has a "reap time" stored in the `next_reap` field of the `kmem_list3` structure inside the cache descriptor (see the section "Cache Descriptor" in Chapter 8); if `jiffies` is still smaller than `next_reap`, it continues with the next cache in the list.

    d. Sets the next "reap time" in the `next_reap` field to a value four seconds from the current time.

e. In multiprocessor systems, the function drains the slab shared cache (see the section "Local Caches of Free Slab Objects" in Chapter 8); this operation could cause new slabs to become free.

f. If a new slab has been recently added to the cache—that is, if the `free_touched` flag of the `kmem_list3` structure inside the cache descriptor is set—it skips this cache and continues with the next cache in the list.

g. Computes according to a heuristic formula the number of slabs to be freed. Basically, this number depends on the upper limit of free objects in the cache and on the number of objects packed into a single slab.

h. Repeatedly invokes `slab_destroy()` on the slabs included in the list of free slabs of the cache, until the list is empty or the target number of free slab has been reached.

i. Invokes `cond_resched()` to check the `TIF_NEED_RESCHED` flag of the current process and to invoke `schedule()`, if the flag is set.

3. Releases the `cache_chain_sem` semaphore.

4. Invokes `schedule_delayed_work()` to schedule the next invocation of the function, and terminates.

The Out of Memory Killer

Despite the PFRA effort to keep a reserve of free page frames, it is possible for the pressure on the virtual memory subsystem to become so high that all available memory becomes exhausted. This situation could quickly induce a freeze of every activity in the system: the kernel keeps trying to free memory in order to satisfy some urgent request, but it does not succeed because the swap areas are full and all disk caches have already been shrunken. As a consequence, no process can proceed with its execution, thus no process will eventually free up the page frames that it owns.

To cope with this dramatic situation, the PFRA makes use of a so-called *out of memory (OOM) killer*, which selects a process in the system and abruptly kills it to free its page frames. The OOM killer is like a surgeon that amputates the limb of a man to save his life: losing a limb is not a nice thing, but sometimes there is nothing better to do.

The `out_of_memory()` function is invoked by `__alloc_pages()` when the free memory is very low and the PFRA has not succeeded in reclaiming any page frames (see the section "The Zone Allocator" in Chapter 8). The function invokes `select_bad_process()` to select a victim among the existing processes, then invokes `oom_kill_process()` to perform the sacrifice.

Of course, `select_bad_process()` does not simply pick a process at random. The selected process should satisfy several requisites:

• The victim should own a large number of page frames, so that the amount of memory that can be freed is significant. (As a countermeasure against the

"fork-bomb" processes, the function considers the amount of memory eaten by all children owned by the parent, too.)

- Killing the victim should lose a small amount of work—it is not a good idea to kill a batch process that has been working for hours or days.

- The victim should be a low static priority process—the users tend to assign lower priorities to less important processes.

- The victim should not be a process with root privileges—they usually perform important tasks.

- The victim should not directly access hardware devices (such as the X Window server), because the hardware could be left in an unpredictable state.

- The victim cannot be *swapper* (process 0), *init* (process 1), or any other kernel thread.

The `select_bad_process()` function scans every process in the system, uses an empirical formula to compute from the above rules a value that denotes how good selecting that process is, and returns the process descriptor address of the "best" candidate for eviction. Then, the `out_of_memory()` function invokes `oom_kill_process()` to send a deadly signal—usually `SIGKILL`; see Chapter 11—either to a child of that process or, if this is not possible, to the process itself. The `oom_kill_process()` function also kills all clones that share the same memory descriptor with the selected victim.

The Swap Token

As you might have realized while reading this chapter, the Linux VM subsystem—and particularly the PFRA—is so complex a piece of code that is quite hard to predict its behavior with an arbitrary workload. There are cases, moreover, in which the VM subsystem exhibits pathological behaviors. An example is the so-called *swap thrashing* phenomenon: essentially, when the system is short of free memory, the PFRA tries hard to free memory by writing pages to disk and stealing the underlying page frames from some processes; at the same time, however, these processes want to proceed with their executions, hence they try hard to access their pages. As a consequence, the kernel assigns to the processes the page frames just freed by the PFRA and reads their contents from disk. The net result is that pages are continuously written to and read back from the disk; most of the time is spent accessing the disk, hence no process makes substantial progress towards its termination.

To mitigate the likelihood of swap thrashing, a technique proposed by Jiang and Zhang in 2004 has been implemented in the kernel version 2.6.9: essentially, a so-called *swap token* is assigned to a single process in the system; the token exempts the process from the page frame reclaiming, so the process can make substantial progress and, hopefully, terminate even when memory is scarce.

The swap token is implemented as a `swap_token_mm` memory descriptor pointer. When a process owns the swap token, `swap_token_mm` is set to the address of the process's memory descriptor.

Immunity from page frame reclaiming is granted in an elegant and simple way. As we have seen in the section "The Least Recently Used (LRU) Lists," a page is moved from the active to the inactive list only if it was not recently referenced. The check is done by the page_referenced() function, which honors the swap token and returns 1 (referenced) if the page belongs to a memory region of the process that owns the swap token. Actually, in a couple of cases the swap token is not considered: when the PFRA is executing on behalf of the process that owns the swap token, and when the PFRA has reached the hardest priority level in page frame reclaiming (level 0).

The grab_swap_token() function determines whether the swap token should be assigned to the current process. It is invoked on each major page fault, namely on just two occasions:

- When the filemap_nopage() function discovers that the required page is not in the page cache (see the section "Demand Paging for Memory Mapping" in Chapter 16).
- When the do_swap_page() function has read a new page from a swap area (see the section "Swapping in Pages" later in this chapter).

The grab_swap_token() function makes some checks before assigning the token. In particular, the token is granted if all of the following conditions hold:

- At least two seconds have elapsed since the last invocation of grab_swap_token().
- The current token-holding process has not raised a major page fault since the last execution of grab_swap_token(), or has been holding the token since at least swap_token_default_timeout ticks.
- The swap token has not been recently assigned to the current process.

The token holding time should ideally be rather long, even in the order of minutes, because the goal is to allow a process to finish its execution. In Linux 2.6.11 the token holding time is set by default to a very low value, namely one tick. However, the system administrator can tune the value of the swap_token_default_timeout variable by writing in the */proc/sys/vm/swap_token_default_timeout* file or by issuing a proper sysctl() system call.

When a process is killed, the kernel checks whether that process was holding the swap token and, if so, releases it; this is done by the mmput() function (see the section "The Memory Descriptor" in Chapter 9).

Swapping

Swapping has been introduced to offer a backup on disk for unmapped pages. We know from the previous discussion that there are three kinds of pages that must be handled by the swapping subsystem:

- Pages that belong to an anonymous memory region of a process (User Mode stack or heap)
- Dirty pages that belong to a private memory mapping of a process
- Pages that belong to an IPC shared memory region (see the section "IPC Shared Memory" in Chapter 19)

Like demand paging, swapping must be transparent to programs. In other words, no special instruction related to swapping needs to be inserted into the code. To understand how this can be done, recall from the section "Regular Paging" in Chapter 2 that each Page Table entry includes a `Present` flag. The kernel exploits this flag to signal that a page belonging to a process address space has been swapped out. Besides that flag, Linux also takes advantage of the remaining bits of the Page Table entry to store into them a "swapped-out page identifier" that encodes the location of the swapped-out page on disk. When a Page Fault exception occurs, the corresponding exception handler can detect that the page is not present in RAM and invoke the function that swaps in the missing page from disk.

The main features of the swapping subsystem can be summarized as follows:

- Set up "swap areas" on disk to store pages that do not have a disk image.
- Manage the space on swap areas allocating and freeing "page slots" as the need occurs.
- Provide functions both to "swap out" pages from RAM into a swap area and to "swap in" pages from a swap area into RAM.
- Make use of "swapped-out page identifiers" in the Page Table entries of pages that are currently swapped out to keep track of the positions of data in the swap areas.

To sum up, swapping is the crowning feature of page frame reclaiming. If we want to be sure that all the page frames obtained by a process, and not only those containing pages that have an image on disk, can be reclaimed at will by the PFRA, then swapping has to be used. Of course, you might turn off swapping by using the *swapoff* command; in this case, however, disk thrashing is likely to occur sooner when the system load increases.

We should also mention that swapping can be used to expand the memory address space that is effectively usable by the User Mode processes. In fact, large swap areas allow the kernel to launch several demanding applications whose total memory requests exceed the amount of physical RAM installed in the system. However, simulation of RAM is not like RAM in terms of performance. Every access by a process to a page that is currently swapped out is of several orders of magnitude longer than an access to a page in RAM. In short, if performance is of great importance, swapping should be used only as a last resort; adding RAM chips still remains the best solution to cope with increasing computing needs.

Swap Area

The pages swapped out from memory are stored in a *swap area*, which may be implemented either as a disk partition of its own or as a file included in a larger partition. Several different swap areas may be defined, up to a maximum number specified by the `MAX_SWAPFILES` macro (usually set to 32).

Having multiple swap areas allows a system administrator to spread a lot of swap space among several disks so that the hardware can act on them concurrently; it also lets swap space be increased at runtime without rebooting the system.

Each swap area consists of a sequence of *page slots*: 4,096-byte blocks used to contain a swapped-out page. The first page slot of a swap area is used to persistently store some information about the swap area; its format is described by the `swap_header` union composed of two structures, `info` and `magic`. The `magic` structure provides a string that marks part of the disk unambiguously as a swap area; it consists of just one field, `magic.magic`, which contains a 10-character "magic" string. The `magic` structure essentially allows the kernel to unambiguously identify a file or a partition as a swap area; the text of the string, namely "SWAPSPACE2," is always located at the end of the first page slot.

The `info` structure includes the following fields:

`bootbits`
> Not used by the swapping algorithm; this field corresponds to the first 1,024 bytes of the swap area, which may store partition data, disk labels, and so on.

`version`
> Swapping algorithm version.

`last_page`
> Last page slot that is effectively usable.

`nr_badpages`
> Number of defective page slots.

`padding[125]`
> Padding bytes.

`badpages[1]`
> Up to 637 numbers specifying the location of defective page slots.

Creating and activating a swap area

The data stored in a swap area is meaningful as long as the system is on. When the system is switched off, all processes are killed, so the data stored by processes in swap areas is discarded. For this reason, swap areas contain very little control information: essentially, the swap area type and the list of defective page slots. This control information easily fits in a single 4 KB page.

Usually, the system administrator creates a swap partition when creating the other partitions on the Linux system, and then uses the *mkswap* command to set up the

disk area as a new swap area. That command initializes the fields just described within the first page slot. Because the disk may include some bad blocks, the program also examines all other page slots to locate the defective ones. But executing the *mkswap* command leaves the swap area in an inactive state. Each swap area can be activated in a script file at system boot or dynamically after the system is running.

Each swap area consists of one or more *swap extents*, each of which is represented by a `swap_extent` descriptor. Each extent corresponds to a group of pages—or more accurately, page slots—that are physically adjacent on disk. Hence, the `swap_extent` descriptor includes the index of the first page of the extent in the swap area, the length in pages of the extent, and the starting disk sector number of the extent. An ordered list of the extents that compose a swap area is created when activating the swap area itself. A swap area stored in a disk partition is composed of just one extent; conversely, a swap area stored in a regular file can be composed of several extents, because the filesystem may not have allocated the whole file in contiguous blocks on disk.

How to distribute pages in the swap areas

When swapping out, the kernel tries to store pages in contiguous page slots to minimize disk seek time when accessing the swap area; this is an important element of an efficient swapping algorithm.

However, if more than one swap area is used, things become more complicated. Faster swap areas—swap areas stored in faster disks—get a higher priority. When looking for a free slot, the search starts in the swap area that has the highest priority. If there are several of them, swap areas of the same priority are cyclically selected to avoid overloading one of them. If no free slot is found in the swap areas that have the highest priority, the search continues in the swap areas that have a priority next to the highest one, and so on.

Swap Area Descriptor

Each active swap area has its own `swap_info_struct` descriptor in memory. The fields of the descriptor are illustrated in Table 17-3.

Table 17-3. Fields of a swap area descriptor

Type	Field	Description
unsigned int	flags	Swap area flags
spinlock_t	sdev_lock	Spin lock protecting the swap area
struct file *	swap_file	Pointer to the file object of the regular file or device file that stores the swap area
struct block_device *	bdev	Descriptor of the block device containing the swap area
struct list head	extent_list	Head of the list of extents that compose the swap area

Table 17-3. Fields of a swap area descriptor (continued)

Type	Field	Description
int	nr_extents	Number of extents composing the swap area
struct swap_extent *	curr_swap_extent	Pointer to the most recently used extent descriptor
unsigned int	old_block_size	Natural block size of the partition containing the swap area
unsigned short *	swap_map	Pointer to an array of counters, one for each swap area page slot
unsigned int	lowest_bit	First page slot to be scanned when searching for a free one
unsigned int	highest_bit	Last page slot to be scanned when searching for a free one
unsigned int	cluster_next	Next page slot to be scanned when searching for a free one
unsigned int	cluster_nr	Number of free page slot allocations before restarting from the beginning
int	prio	Swap area priority
int	pages	Number of usable page slots
unsigned long	max	Size of swap area in pages
unsigned long	inuse_pages	Number of used page slots in the swap area
int	next	Pointer to next swap area descriptor

The flags field includes three overlapping subfields:

SWP_USED
> 1 if the swap area is active; 0 if it is inactive.

SWP_WRITEOK
> 1 if it is possible to write into the swap area; 0 if the swap area is read-only (it is being activated or inactivated).

SWP_ACTIVE
> This 2-bit field is actually the combination of SWP_USED and SWP_WRITEOK; the flag is set when both the previous flags are set.

The swap_map field points to an array of counters, one for each swap area page slot. If the counter is equal to 0, the page slot is free; if it is positive, the page slot is filled with a swapped-out page. Essentially, the page slot counter denotes the number of processes that share the swapped-out page. If the counter has the value SWP_MAP_MAX (equal to 32,767), the page stored in the page slot is "permanent" and cannot be removed from the corresponding slot. If the counter has the value SWP_MAP_BAD (equal to 32,768), the page slot is considered defective, and thus unusable.*

---

* "Permanent" page slots protect against overflows of swap_map counters. Without them, valid page slots could become "defective" if they are referenced too many times, thus leading to data losses. However, no one really expects that a page slot counter could reach the value 32,768. It's just a "belt and suspenders" approach.

The `prio` field is a signed integer that denotes the order in which the swap subsystem should consider each swap area.

The `sdev_lock` field is a spin lock that protects the swap area's data structures—chiefly, the swap descriptor—against concurrent accesses in SMP systems.

The `swap_info` array includes `MAX_SWAPFILES` swap area descriptors. Only the areas whose `SWP_USED` flags are set are used, because they are the activated areas. Figure 17-6 illustrates the `swap_info` array, one swap area, and the corresponding array of counters.
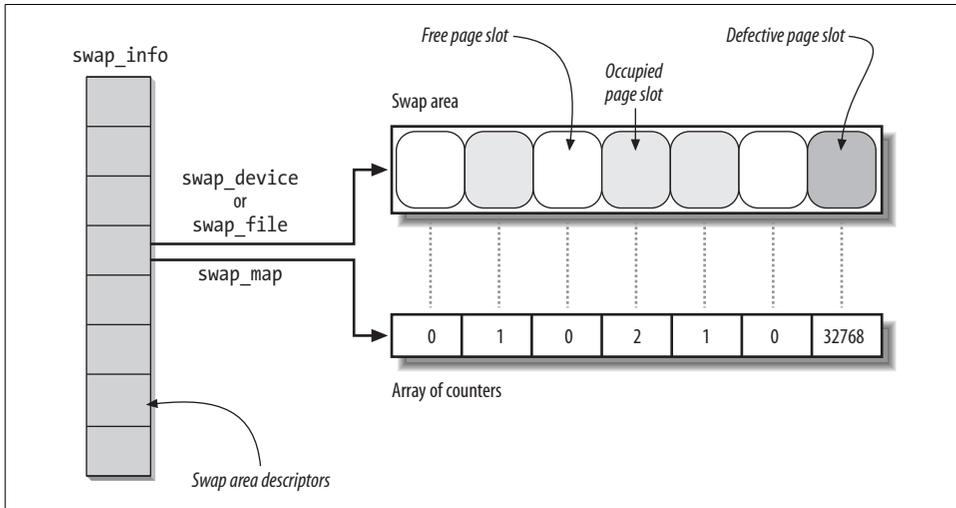
Figure 17-6. Swap area data structures

The `nr_swapfiles` variable stores the index of the last array element that contains, or that has contained, a used swap area descriptor. Despite its name, the variable *does not* contain the number of active swap areas.

Descriptors of active swap areas are also inserted into a list sorted by the swap area priority. The list is implemented through the `next` field of the swap area descriptor, which stores the index of the next descriptor in the `swap_info` array. This use of the field as an index is different from most fields with the name `next`, which are usually pointers.

The `swap_list` variable, of type `swap_list_t`, includes the following fields:

head
    Index in the `swap_info` array of the first list element.

next
    Index in the `swap_info` array of the descriptor of the next swap area to be selected for swapping out pages. This field is used to implement a Round Robin algorithm among maximum-priority swap areas with free slots.

The swaplock spin lock protects the list against concurrent accesses in multiprocessor systems.

The max field of the swap area descriptor stores the size of the swap area in pages, while the pages field stores the number of usable page slots. These numbers differ because pages does not take the first page slot and the defective page slots into consideration.

Finally, the nr_swap_pages variable contains the number of available (free and nondefective) page slots in all active swap areas, while the total_swap_pages variable contains the total number of nondefective page slots.

Swapped-Out Page Identifier

A swapped-out page is uniquely identified quite simply by specifying the index of the swap area in the swap_info array and the page slot index inside the swap area. Because the first page (with index 0) of the swap area is reserved for the swap_header union discussed earlier, the first useful page slot has index 1. The format of a *swapped-out page identifier* is illustrated in Figure 17-7.
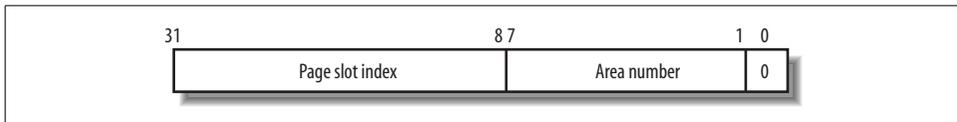
Figure 17-7. Swapped-out page identifier

The swp_entry(type,offset) function constructs a swapped-out page identifier from the swap area index type and the page slot index offset. Conversely, the swp_type and swp_offset functions extract from a swapped-out page identifier the swap area index and the page slot index, respectively.

When a page is swapped out, its identifier is inserted as the page's entry into the Page Table so the page can be found again when needed. Notice that the least-significant bit of such an identifier, which corresponds to the Present flag, is always cleared to denote the fact that the page is not currently in RAM. However, at least one of the remaining 31 bits has to be set because no page is ever stored in slot 0 of swap area 0. It is therefore possible to identify three different cases from the value of a Page Table entry:

Null entry
>   The page does not belong to the process address space, or the underlying page frame has not yet been assigned to the process (demand paging).

First 31 most-significant bits not all equal to 0, last bit equal to 0
>   The page is currently swapped out.

Least-significant bit equal to 1
>   The page is contained in RAM.

The maximum size of a swap area is determined by the number of bits available to identify a slot. On the 80×86 architecture, the 24 bits available limit the size of a swap area to 2^{24} slots (that is, to 64 GB).

Because a page may belong to the address spaces of several processes (see the earlier section "Reverse Mapping"), it may be swapped out from the address space of one process and still remain in main memory; therefore, it is possible to swap out the same page several times. A page is physically swapped out and stored just once, of course, but each subsequent attempt to swap it out increases the swap_map counter.

The swap_duplicate() function is usually invoked while trying to swap out an already swapped-out page. It simply verifies that the swapped-out page identifier passed as its parameter is valid and increases the corresponding swap_map counter. More precisely, it performs the following actions:

1. Uses the swp_type and swp_offset functions to extract the swap area number and the page slot index from the parameter.

2. Checks whether the swap area number identified is active; if not, it returns 0 (invalid identifier).

3. Checks whether the page slot is valid and not free (its swap_map counter is greater than 0 and less than SWAP_MAP_BAD); if not, it returns 0 (invalid identifier).

4. Otherwise, the swapped-out page identifier locates a valid page. Increases the swap_map counter of the page slot if it has not already reached the value SWAP_MAP_MAX.

5. Returns 1 (valid identifier).

Activating and Deactivating a Swap Area

Once a swap area is initialized, the superuser (or, more precisely, every user having the CAP_SYS_ADMIN capability, as described in the section "Process Credentials and Capabilities" in Chapter 20) may use the *swapon* and *swapoff* programs to activate and deactivate the swap area, respectively. These programs use the swapon( ) and swapoff( ) system calls; we'll briefly sketch out the corresponding service routines.

The sys_swapon( ) service routine

The sys_swapon( ) service routine receives the following as its parameters:

specialfile
> This parameter points to the pathname (in the User Mode address space) of the device file (partition) or plain file used to implement the swap area.

swap_flags
> This parameter consists of a single SWAP_FLAG_PREFER bit plus 31 bits of priority of the swap area (these bits are significant only if the SWAP_FLAG_PREFER bit is on).

The function checks the fields of the `swap_header` union that was put in the first slot when the swap area was created. The function performs these main steps:

1. Checks that the current process has the `CAP_SYS_ADMIN` capability.

2. Looks in the first `nr_swapfiles` components of the `swap_info` array of swap area descriptors for the first descriptor having the `SWP_USED` flag cleared, meaning that the corresponding swap area is inactive. If an inactive swap area is found, it goes to step 4.

3. The new swap area array index is equal to `nr_swapfiles`: it checks that the number of bits reserved for the swap area index is sufficiently large to encode the new index; if not, returns an error code; otherwise, it increases by one the value of `nr_swapfiles`.

4. An index of an unused swap area has been found: it initializes the descriptor's fields; in particular, it sets `flags` to `SWP_USED`, and sets `lowest_bit` and `highest_bit` to 0.

5. If the `swap_flags` parameter specifies a priority for the new swap area, the function sets the `prio` field of the descriptor. Otherwise, it initializes the field to one less than the lowest priority among all active swap areas (thus assuming that the last activated swap area is on the slowest block device). If no other swap areas are already active, the function assigns the value –1.

6. Copies the string pointed to by the `specialfile` parameter from the User Mode address space.

7. Invokes `filp_open()` to open the file specified by the `specialfile` parameter (see the section "The open() System Call" in Chapter 12).

8. Stores the addresses of the file object returned by `filp_open()` in the `swap_file` field of the swap area descriptor.

9. Makes sure that the swap area is not already activated by looking at the other active swap areas in `swap_info`. This is done by checking the addresses of the `address_space` objects stored in the `swap_file->f_mapping` field of the swap area descriptors. If the swap area is already active, it returns an error code.

10. If the `specialfile` parameter identifies a block device file, it performs the following substeps:

    a. Invokes `bd_claim()` to set the swapping subsystem as the holder of the block device (see the section "Block Devices" in Chapter 14). If the block device already has a holder, it returns an error code.

    b. Stores the address of the `block_device` descriptor in the `bdev` field of the swap area descriptor.

    c. Stores the current block size of the device in the `old_block_size` field of the swap area descriptor, then sets the block size of the device to 4,096 bytes (the page size).

---

11. If the `specialfile` parameter identifies a regular file, it performs the following substeps:

    a. Checks the `S_SWAPFILE` field of the `i_flags` field of the file's inode. If this flag is set, it returns an error code because the file is already being used as a swap area.

    b. Stores the descriptor address of the block device containing the file in the `bdev` field of the swap area descriptor.

12. Reads the `swap_header` descriptor stored in slot 0 of the swap area. To that end, it invokes `read_cache_page()` passing as parameters the `address_space` object pointed to by `swap_file->f_mapping`, the page index 0, the address of the file's `readpage` method (stored in `swap_file->f_mapping->a_ops->readpage`), and the pointer to the file object `swap_file`. Waits until the page has been read into memory.

13. Checks that the magic string in the last 10 characters of the first page is equal to "SWAPSPACE2." If not, it returns an error code.

14. Initializes the `lowest_bit` and `highest_bit` fields of the swap area descriptor according to the size of the swap area stored in the `info.last_page` field of the `swap_header` union.

15. Invokes `vmalloc()` to create the array of counters associated with the new swap area and stores its address in the `swap_map` field of the swap descriptor. Initializes the elements of the array to 0 or to `SWAP_MAP_BAD`, according to the list of defective page slots stored in the `info.bad_pages` field of the `swap_header` union.

16. Computes the number of useful page slots by accessing the `info.last_page` and `info.nr_badpages` fields in the first page slot, and stores it in the `pages` field of the swap area descriptor. Also sets the `max` field with the total number of pages in the swap area.

17. Builds the `extent_list` list of swap extents for the new swap area (only one if the swap area is a disk partition), and sets properly the `nr_extents` and `curr_swap_extent` fields in the swap area descriptor.

18. Sets the `flags` field of the swap area descriptor to `SWP_ACTIVE`.

19. Updates the `nr_good_pages`, `nr_swap_pages`, and `total_swap_pages` global variables.

20. Inserts the swap area descriptor in the list to which the `swap_list` variable points.

21. Returns 0 (success).

The sys_swapoff( ) service routine

The `sys_swapoff()` service routine deactivates a swap area identified by the parameter `specialfile`. It is much more complex and time-consuming than `sys_swapon()`, since the partition to be deactivated might still contain pages that belong to several

processes. The function is thus forced to scan the swap area and to swap in all existing pages. Because each swap-in requires a new page frame, it might fail if there are no free page frames left. In this case, the function returns an error code. All this is achieved by performing the following major steps:

1. Checks that the current process has the CAP_SYS_ADMIN capability.

2. Copies the string pointed to by the specialfile parameter in kernel space.

3. Invokes filp_open() to open the file referenced by the specialfile parameter; as usual, this function returns the address of a file object.

4. Scans the swap_list list of the swap area descriptor, and compares the address of the file object returned by filp_open() with the addresses stored in the swap_file fields of the active swap area descriptors. If no match is found, an invalid parameter was passed to the function, so it returns an error code.

5. Invokes cap_vm_enough_memory() to check whether there are enough free page frames to swap in all pages stored in the swap area. If not, the swap area cannot be deactivated; it releases the file object and returns an error code. This is only a rough check, but it could save the kernel from a lot of useless disk activity. While performing this check, cap_vm_enough_memory() takes into account the page frames allocated through slab caches having the SLAB_RECLAIM_ACCOUNT flag set (see the section "Interfacing the Slab Allocator with the Zoned Page Frame Allocator" in Chapter 8). The number of such pages, which are considered as reclaimable, is stored in the slab_reclaim_pages variable.

6. Removes the swap area descriptor from the swap_list list.

7. Updates the nr_swap_pages and total_swap_pages variables by subtracting the value in the pages field of the swap area descriptor.

8. Clears the SWP_WRITEOK flag in the flags field of the swap area descriptor; this forbids the PFRA from swapping out more pages in the swap area.

9. Invokes try_to_unuse() (see below) to successively force all pages left in the swap area into RAM and to correspondingly update the Page Tables of the processes that use these pages. While executing this function, the current process, which is executing the *swapoff* command, has the PF_SWAPOFF flag set. Setting this flag has just one consequence: in case of a dramatic shortage of page frames, the select_bad_process() function will be forced to select and kill this process! (See the section "The Out of Memory Killer" earlier in this chapter.)

10. Waits until the block device driver that contains the swap area is unplugged (see the section "Activating the Block Device Driver" in Chapter 14). In this way, the reading requests submitted by try_to_unuse() will be handled by the driver before the swap area is deactivated.

11. If try_to_unuse() fails in allocating all requested page frames, the swap area cannot be deactivated. Therefore, the function executes the following substeps:

a. Reinserts the swap area descriptor in the `swap_list` list and sets its `flags` field to `SWP_WRITEOK`.

b. Restores the original contents of the `nr_swap_pages` and `total_swap_pages` variables by adding the value in the `pages` field of the swap area descriptor.

c. Invokes `filp_close()` to close the file opened in step 3 (see the section "The close() System Call" in Chapter 12), and returns an error code.

12. Otherwise, all used page slots have been successfully transferred to RAM. Therefore, the function executes the following substeps:

a. Releases the memory areas used to store the `swap_map` array and the extent descriptors.

b. If the swap area is stored in a disk partition, it restores the block size to its original value, which is stored in the `old_block_size` field of the swap area descriptor; moreover, it invokes the `bd_release()` function so that the swap subsystem no longer holds the block device (see step 10a in the description of `sys_swapon()`).

c. If the swap area is stored in a regular file, it clears the `S_SWAPFILE` flag of the file's inode.

d. Invokes `filp_close()` twice, the first time on the `swap_file` file object, the second time on the object returned by `filp_open()` in step 3.

e. Returns 0 (success).

The try_to_unuse( ) function

The `try_to_unuse()` function acts on an index parameter that identifies the swap area to be emptied; it swaps in pages and updates all the Page Tables of processes that have swapped out pages in this swap area. To that end, the function visits the address spaces of all kernel threads and processes, starting with the `init_mm` memory descriptor that is used as a marker. It is a time-consuming function that runs mostly with the interrupts enabled. Synchronization with other processes is therefore critical.

The `try_to_unuse()` function scans the `swap_map` array of the swap area. When the function finds a in-use page slot, it first swaps in the page, and then starts looking for the processes that reference the page. The ordering of these two operations is crucial to avoid race conditions. While the I/O data transfer is ongoing, the page is locked, so no process can access it. Once the I/O data transfer completes, the page is locked again by `try_to_unuse()`, so it cannot be swapped out again by another kernel control path. Race conditions are also avoided because each process looks up the page cache before starting a swap-in or swap-out operation (see the later section "The Swap Cache"). Finally, the swap area considered by `try_to_unuse()` is marked as nonwritable (`SWP_WRITEOK` flag is not set), so no process can perform a swap-out on a page slot of this area.

However, `try_to_unuse()` might be forced to scan the `swap_map` array of usage counters of the swap area several times. This is because memory regions that contain references to swapped-out pages might disappear during one scan and later reappear in the process lists.

For instance, recall the description of the `do_munmap()` function (in the section "Releasing a Linear Address Interval" in Chapter 9): whenever a process releases an interval of linear addresses, `do_munmap()` removes from the process list all memory regions that include the affected linear addresses; later, the function reinserts the memory regions that have been only partially unmapped in the process list. `do_munmap()` takes care of freeing the swapped-out pages that belong to the interval of released linear addresses. It commendably doesn't free the swapped-out pages that belong to the memory regions that have to be reinserted in the process list.

Hence, `try_to_unuse()` might fail in finding a process that references a given page slot because the corresponding memory region is temporarily not included in the process list. To cope with this fact, `try_to_unuse()` keeps scanning the `swap_map` array until all reference counters are null. Eventually, the ghost memory regions referencing the swapped-out pages will reappear in the process lists, so `try_to_unuse()` will succeed in freeing all page slots.

Let's describe now the major operations executed by `try_to_unuse()`. It executes a continuous loop on the reference counters in the `swap_map` array of the swap area passed as its parameter. This loop is interrupted and the function returns an error code if the current process receives a signal. For each reference counter, the function performs the following steps:

1. If the counter is equal to 0 (no page is stored there) or to `SWAP_MAP_BAD`, it continues with the next page slot.

2. Otherwise, it invokes the `read_swap_cache_async()` function (see the section "Swapping in Pages" later in this chapter) to swap in the page. This consists of allocating, if necessary, a new page frame, filling it with the data stored in the page slot, and putting the page in the swap cache.

3. Waits until the new page has been properly updated from disk and locks it.

4. While the function was executing the previous step, the process could have been suspended. Therefore, it checks again whether the reference counter of the page slot is null; if so, this swap page has been freed by another kernel control path, so the function continues with the next page slot.

5. Invokes `unuse_process()` on every memory descriptor in the doubly linked list whose head is `init_mm` (see the section "The Memory Descriptor" in Chapter 9). This time-consuming function scans all Page Table entries of the process that owns the memory descriptor, and replaces each occurrence of the swapped-out page identifier with the physical address of the page frame. To reflect this move, the function also decreases the page slot counter in the `swap_map` array (unless it is equal to `SWAP_MAP_MAX`) and increases the usage counter of the page frame.

6. Invokes `shmem_unuse()` to check whether the swapped-out page is used as an IPC shared memory resource and to properly handle that case (see the section "IPC Shared Memory" in Chapter 19).

7. Checks the value of the reference counter of the page. If it is equal to `SWAP_MAP_MAX`, the page slot is "permanent." To free it, it forces the value 1 into the reference counter.

8. The swap cache might own the page as well (it contributes to the value of the reference counter). If the page belongs to the swap cache, it invokes the `swap_writepage()` function to flush its contents to disk (if the page is dirty) and invokes `delete_from_swap_cache()` to remove the page from the swap cache and to decrease its reference counter.

9. Sets the `PG_dirty` flag of the page descriptor, unlocks the page frame, and decreases its reference counter (to undo the increment done in step 5).

10. Checks the `need_resched` field of the current process; if it is set, it invokes `schedule()` to relinquish the CPU. Deactivating a swap area is a long job, and the kernel must ensure that the other processes in the system still continue to execute. The `try_to_unuse()` function continues from this step whenever the process is selected again by the scheduler.

11. Proceeds with the next page slot, starting at step 1.

The function continues until every reference counter in the `swap_map` array is null. Recall that even if the function starts examining the next page slot, the reference counter of the previous page slot could still be positive. In fact, a "ghost" process could still reference the page, typically because some memory regions have been temporarily removed from the process list scanned in step 5. Eventually, `try_to_unuse()` catches every reference. In the meantime, however, the page is no longer in the swap cache, it is unlocked, and a copy is still included in the page slot of the swap area being deactivated.

One might expect that this situation could lead to data loss. For instance, suppose that some "ghost" process accesses the page slot and starts swapping the page in. Because the page is no longer in the swap cache, the process fills a new page frame with the data read from disk. However, this page frame would be different from the page frames owned by the processes that are supposed to share the page with the "ghost" process.

This problem does not arise when deactivating a swap area, because interference from a ghost process could happen only if a swapped-out page belongs to a private anonymous memory mapping.[*] In this case, the page frame is handled by means of the Copy On Write mechanism described in Chapter 9, so it is perfectly legal to

---

[*] Actually, the page might also belong to an IPC shared memory region; Chapter 19 has a discussion of this case.

assign different page frames to the processes that reference the page. However, the try_to_unuse() function marks the page as "dirty" (step 9); otherwise, the shrink_list() function might later drop the page from the Page Table of some process without saving it in an another swap area (see the later section "Swapping Out Pages").

Allocating and Releasing a Page Slot

As we will see later, when freeing memory, the kernel swaps out many pages in a short period of time. It is therefore important to try to store these pages in contiguous slots to minimize disk seek time when accessing the swap area.

A first approach to an algorithm that searches for a free slot could choose one of two simplistic, rather extreme strategies:

- Always start from the beginning of the swap area. This approach may increase the average seek time during swap-out operations, because free page slots may be scattered far away from one another.

- Always start from the last allocated page slot. This approach increases the average seek time during swap-in operations if the swap area is mostly free (as is usually the case), because the handful of occupied page slots may be scattered far away from one another.

Linux adopts a hybrid approach. It always starts from the last allocated page slot unless one of these conditions occurs:

- The end of the swap area is reached.

- SWAPFILE_CLUSTER (usually 256) free page slots were allocated after the last restart from the beginning of the swap area.

The cluster_nr field in the swap_info_struct descriptor stores the number of free page slots allocated. This field is reset to 0 when the function restarts allocation from the beginning of the swap area. The cluster_next field stores the index of the first page slot to be examined in the next allocation.[*]

To speed up the search for free page slots, the kernel keeps the lowest_bit and highest_bit fields of each swap area descriptor up-to-date. These fields specify the first and the last page slots that could be free; in other words, every page slot below lowest_bit and above highest_bit is known to be occupied.

The scan_swap_map() function

The scan_swap_map() function is used to find a free page slot in a given swap area. It acts on a single parameter, which points to a swap area descriptor and returns the

---

[*] As you may have noticed, the names of Linux data structures are not always appropriate. In this case, the kernel does not really "cluster" page slots of a swap area.

index of a free page slot. It returns 0 if the swap area does not contain any free slots. The function performs the following steps:

1. It tries first to use the current cluster. If the `cluster_nr` field of the swap area descriptor is positive, it scans the `swap_map` array of counters starting from the element at index `cluster_next` and looks for a null entry. If a null entry is found, it decreases the `cluster_nr` field and goes to step 4.

2. If this point is reached, either the `cluster_nr` field is null or the search starting from `cluster_next` didn't find a null entry in the `swap_map` array. It is time to try the second stage of the hybrid search. The function reinitializes `cluster_nr` to `SWAPFILE_CLUSTER` and restarts scanning the array from the `lowest_bit` index trying to find a group of `SWAPFILE_CLUSTER` free page slots. If such a group is found, it goes to step 4.

3. No group of `SWAPFILE_CLUSTER` free page slots exists. The function restarts scanning the array from the `lowest_bit` index trying to find a single free page slot. If no null entry is found, it sets the `lowest_bit` field to the maximum index in the array, the `highest_bit` field to 0, and returns 0 (the swap area is full).

4. A null entry is found. Puts the value 1 in the entry, decreases `nr_swap_pages`, updates the `lowest_bit` and `highest_bit` fields if necessary, increases the `inuse_pages` field by one, and sets the `cluster_next` field to the index of the page slot just allocated plus 1.

5. Returns the index of the allocated page slot.

The get_swap_page( ) function

The get_swap_page( ) function is used to find a free page slot by searching all the active swap areas. The function, which returns the swapped-out page identifier of a newly allocated page slot or 0 if all swap areas are filled, takes into consideration the different priorities of the active swap areas.

Two passes are done in order to minimize runtime when it's easy to find a page slot. The first pass is partial and applies only to areas that have a single priority; the function searches such areas in a Round Robin fashion for a free slot. If no free page slot is found, a second pass is made starting from the beginning of the swap area list; during this second pass, all swap areas are examined. More precisely, the function performs the following steps:

1. If `nr_swap_pages` is null or if there are no active swap areas, it returns 0.

2. Starts by considering the swap area pointed to by `swap_list.next` (recall that the swap area list is sorted by decreasing priorities).

3. If the swap area is active, it invokes scan_swap_map( ) to allocate a free page slot. If scan_swap_map( ) returns a page slot index, the function's job is essentially done, but it must prepare for its next invocation. Thus, it updates `swap_list.next` to point to the next swap area in the swap area list, if the latter has the same priority

(thus continuing the round-robin use of these swap areas). If the next swap area does not have the same priority as the current one, the function sets `swap_list.next` to the first swap area in the list (so that the next search will start with the swap areas that have the highest priority). The function finishes by returning the swapped-out page identifier corresponding to the page slot just allocated.

4. Either the swap area is not writable, or it does not have free page slots. If the next swap area in the swap area list has the same priority as the current one, the function makes it the current one and goes to step 3.

5. At this point, the next swap area in the swap area list has a lower priority than the previous one. The next step depends on which of the two passes the function is performing.

   a. If this is the first (partial) pass, it considers the first swap area in the list and goes to step 3, thus starting the second pass.

   b. Otherwise, it checks if there is a next element in the list; if so, it considers it and goes to step 3.

6. At this point the list is completely scanned by the second pass and no free page slot has been found; it returns 0.

The swap_free( ) function

The `swap_free( )` function is invoked when swapping in a page to decrease the corresponding `swap_map` counter (see Table 17-3). When the counter reaches 0, the page slot becomes free since its identifier is no longer included in any Page Table entry. We'll see in the later section "The Swap Cache," however, that the swap cache counts as an owner of the page slot.

The function acts on a single `entry` parameter that specifies a swapped-out page identifier and performs the following steps:

1. Derives the swap area index and the `offset` page slot index from the `entry` parameter and gets the address of the swap area descriptor.

2. Checks whether the swap area is active and returns right away if it is not.

3. If the `swap_map` counter corresponding to the page slot being freed is smaller than `SWAP_MAP_MAX`, the function decreases it. Recall that entries that have the `SWAP_MAP_MAX` value are considered persistent (undeletable).

4. If the `swap_map` counter becomes 0, the function increases the value of `nr_swap_pages`, decreases the `inuse_pages` field, and updates, if necessary, the `lowest_bit` and `highest_bit` fields of the swap area descriptor.

The Swap Cache

Transferring pages to and from a swap area is an activity that can induce many race conditions. In particular, the swapping subsystem must handle carefully the following cases:

Multiple swap-ins

  Two processes may concurrently try to swap in the same shared anonymous page.

Concurrent swap-ins and swap-outs

  A process may swap-in a page that is being swapped out by the PFRA.

The *swap cache* has been introduced to solve these kinds of synchronization problems. The key rule is that nobody can start a swap-in or swap-out without checking whether the swap cache already includes the affected page. Thanks to the swap cache, concurrent swap operations affecting the same page always act on the same page frame; therefore, the kernel may safely rely on the PG_locked flag of the page descriptor to avoid any race condition.

For example, consider two processes that share the same swapped-out page. When the first process tries to access the page, the kernel starts the swap-in operation. The very first step consists of checking whether the page frame is already included in the swap cache. Let's suppose it isn't: then, the kernel allocates a new page frame and inserts it into the swap cache; next, it starts the I/O operation to read the page's contents from the swap area. Meanwhile, the second process accesses the shared anonymous page. As above, the kernel starts a swap-in operation and checks whether the affected page frame is already included in the swap cache. Now, it is included, thus the kernel simply accesses the page frame descriptor and puts the current process to sleep until the PG_locked flag is cleared, that is, until the I/O data transfer completes.

The swap cache plays a crucial role also when concurrent swap-in and swap-out operations mix up. As explained in the section "Low On Memory Reclaiming" earlier in this chapter, the shrink_list() function starts swapping out an anonymous page only if try_to_unmap() succeeds in removing the page frame from the User Mode Page Tables of all processes that own the page. However, one of these processes may access the page and cause a swap-in while the swap-out write operation is still in progress.

Before being written to disk, each page to be swapped out is stored in the swap cache by shrink_list(). Consider a page P that is shared among two processes, A and B. Initially, the Page Table entries of both processes contain a reference to the page frame, and the page has two owners; this case is illustrated in Figure 17-8(a). When the PFRA selects the page for reclaiming, shrink_list() inserts the page frame in the swap cache. As illustrated in Figure 17-8(b), now the page frame has three owners, while the page slot in the swap area is referenced only by the swap cache. Next, the PFRA invokes try_to_unmap() to remove the references to the page frame from the Page Table of the processes; once this function terminates, the page frame is referenced only by the swap cache, while the page slot is referenced by the two processes and the swap cache, as illustrated in Figure 17-8(c). Let's suppose that, while the page's contents are being written to disk, process B accesses the page—that is, it tries to access a memory cell using a linear address inside the page. Then, the page fault

handler finds the page frame in the swap cache and puts back its physical address in the Page Table entry of process B, as illustrated in Figure 17-8(d). Conversely, if the swap-out operation terminates without concurrent swap-in operations, the `shrink_list()` function removes the page frame from the swap cache and releases the page frame to the Buddy system, as illustrated in Figure 17-8(e).
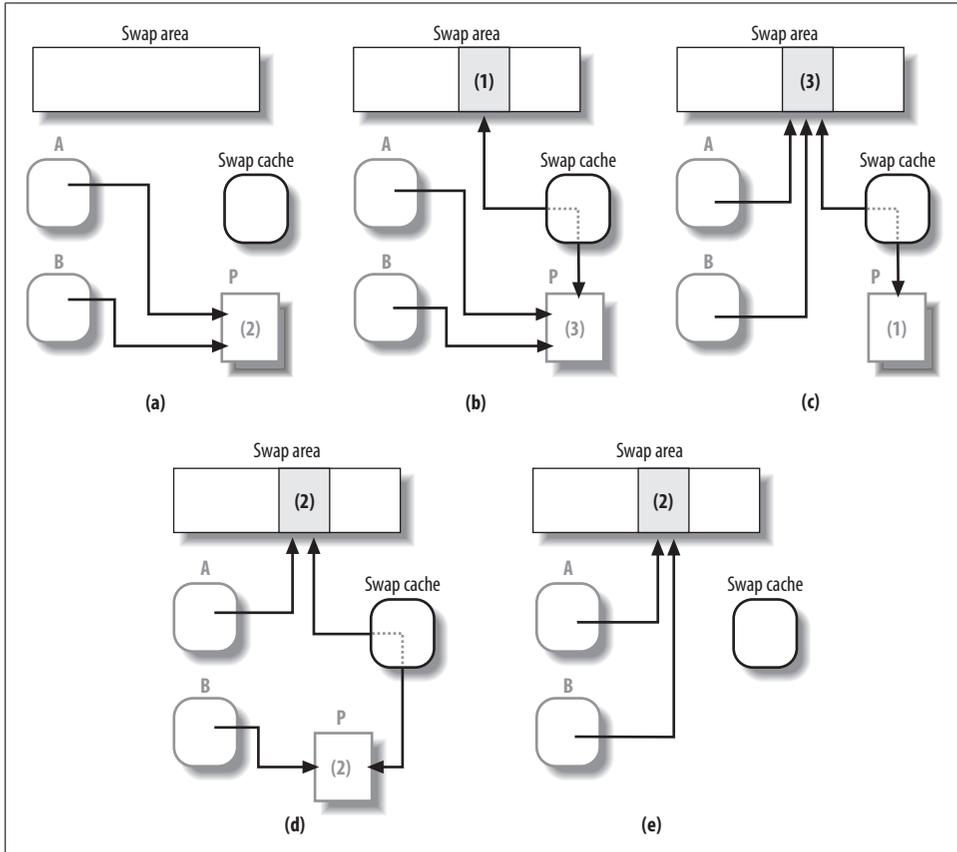
Figure 17-8. The role of the swap cache

You might consider the swap cache as a transit area containing the page descriptors of anonymous pages that are being currently swapped-in or swapped out. When the swap-in or swap-out terminates (in the case of shared anonymous pages, the swap-in or swap-out must have been performed on all the processes that share the page), the page descriptor of the anonymous page may be removed from the swap cache.[*]

---

[*] In some cases, the swap cache improves also the system performance: consider a server daemon that services requests by creating child processes. Under heavy system load, a page can get swapped out from the parent process, and it will never be paged in for the parent process. Without the swap cache, every child process that gets forked off needs to fault that page in from the swap area.

Swap cache implementation

The swap cache is implemented by the page cache data structures and procedures, which are described in the section "The Page Cache" in Chapter 15. Recall that the core of the page cache is a set of radix trees that allows the algorithm to quickly derive the address of a page descriptor from the address of an `address_space` object identifying the owner of the page as well as from an offset value.

Pages in the swap cache are stored as every other page in the page cache, with the following special treatment:

- The `mapping` field of the page descriptor is set to `NULL`.
- The `PG_swapcache` flag of the page descriptor is set.
- The `private` field stores the swapped-out page identifier associated with the page.

Moreover, when the page is put in the swap cache, both the `count` field of the page descriptor and the page slot usage counters are increased, because the swap cache uses both the page frame and the page slot.

Finally, a single `swapper_space` address space is used for all pages in the swap cache, so a single radix tree pointed to by `swapper_space.page_tree` addresses the pages in the swap cache. The `nrpages` field of the `swapper_space` address space stores the number of pages contained in the swap cache.

Swap cache helper functions

The kernel uses several functions to handle the swap cache; they are based mainly on those discussed in the section "The Page Cache" in Chapter 15. We show later how these relatively low-level functions are invoked by higher-level functions to swap pages in and out as needed.

The main functions that handle the swap cache are:

`lookup_swap_cache( )`
> Finds a page in the swap cache through its swapped-out page identifier passed as a parameter and returns the page descriptor address. It returns 0 if the page is not present in the cache. To find the required page, it invokes `radix_tree_lookup( )`, passing as parameters a pointer to `swapper_space.page_tree`—the radix tree used for pages in the swap cache—and the swapped-out page identifier.

`add_to_swap_cache( )`
> Inserts a page into the swap cache. It essentially invokes `swap_duplicate( )` to check whether the page slot passed as a parameter is valid and to increase the page slot usage counter; then, it invokes `radix_tree_insert( )` to insert the page into the cache; finally, it increases the page's reference counter and sets the `PG_swapcache` and `PG_locked` flags.

`__add_to_swap_cache()`
> Similar to `add_to_swap_cache()`, except that the function does not invoke `swap_duplicate()` before inserting the page frame in the swap cache.

`delete_from_swap_cache()`
> Removes a page from the swap cache by invoking `radix_tree_delete()`, decreases the corresponding usage counter in `swap_map`, and decreases the page reference counter.

`free_page_and_swap_cache()`
> Removes a page from the swap cache if no User Mode process besides `current` is referencing the corresponding page slot, and decreases the page's usage counter.

`free_pages_and_swap_cache()`
> Analogous to `free_page_and_swap_cache()`, but operates on a set of pages.

`free_swap_and_cache()`
> Frees a swap entry, and checks whether the page referenced by the entry is in the swap cache. If either no User Mode process, besides `current`, is referencing the page or more than 50% of the swap entries are busy, the function removes the page from the swap cache.

Swapping Out Pages

We have seen in the section "Low On Memory Reclaiming" earlier in this chapter how the PFRA determines whether a given anonymous page should be swapped out. In this section we show how the kernel performs a swap-out.

Inserting the page frame in the swap cache

The first step of a swap-out operation consists of preparing the swap cache. If the `shrink_list()` function determines that a page is anonymous (the `PageAnon()` function returns 1) and that the swap cache does not include the corresponding page frame (the `PG_swapcache` flag in the page descriptor is clear), the kernel invokes the `add_to_swap()` function.

The `add_to_swap()` function allocates a new page slot in a swap area and inserts a page frame—whose page descriptor address is passed as its parameter—in the swap cache. Essentially, the function performs the following steps:

1. Invokes `get_swap_page()` to allocate a new page slot; see the section "Allocating and Releasing a Page Slot" earlier in this chapter. Returns 0 in case of failure (for example, no free page slot found).

2. Invokes `__add_to_page_cache()`, passing to it the page slot index, the page descriptor address, and some allocation flags.

3. Sets the `PG_uptodate` and `PG_dirty` flags in the page descriptor, so that the `shrink_list()` function will be forced to write the page to disk (see the next section).

4. Returns 1 (success).

Updating the Page Table entries

Once add_to_swap() terminates, shrink_list() invokes try_to_unmap(), which deter-
mines the address of every User Mode page table entry referring to the anonymous
page and writes into it a swapped-out page identifier; this is described in the section
"Reverse Mapping for Anonymous Pages" earlier in this chapter.

Writing the page into the swap area

The next action to be performed to complete the swap-out consists of writing the
page's contents into the swap area. This I/O transfer is activated by the shrink_list()
function, which checks whether the PG_dirty flag of the page frame is set and conse-
quently executes the pageout() function (see Figure 17-5 earlier in this chapter).

As explained in the section "Low On Memory Reclaiming" earlier in this chapter, the
pageout() function sets up a writeback_control descriptor and invokes the writepage
method of the page's address_space object. The writepage method of the swapper_
state object is implemented by the swap_writepage() function.

The swap_writepage() function, in turn, performs essentially the following steps:

1. Checks whether at least one User Mode process is referencing the page. If not, it
   removes the page from the swap cache and returns 0. This check is necessary
   because a process might race with the PRFA and release a page after the check
   performed by shrink_list().

2. Invokes get_swap_bio() to allocate and initialize a bio descriptor (see the section
   "The Bio Structure" in Chapter 14). The function derives the address of the swap
   area descriptor from the swapped-out page identifier; then, it walks the swap
   extent lists to determine the initial disk sector of the page slot. The bio descrip-
   tor will include a request for a single page of data (the page slot); the completion
   method is set to the end_swap_bio_write() function.

3. Sets the PG_writeback flag in the page descriptor and the writeback tags in the
   swap cache's radix tree (see the section "The Tags of the Radix Tree" in
   Chapter 15). Moreover, the function resets the PG_locked flag.

4. Invokes submit_bio(), passing to it the WRITE command and the bio descriptor
   address.

5. Returns 0.

Once the I/O data transfer terminates, the end_swap_bio_write() function is exe-
cuted. Essentially, this function wakes up any process waiting until the PG_writeback
flag of the page is cleared, clears the PG_writeback flag and the corresponding tags in
the radix tree, and releases the bio descriptor used for the I/O transfer.

Removing the page frame from the swap cache

The last step of the swap-out operation is performed once more by shrink_list(): if
it verifies that no process has tried to access the page frame while doing the I/O data

transfer, it essentially invokes `delete_from_swap_cache()` to remove the page frame from the swap cache. Because the swap cache was the only owner of the page, the page frame is released to the buddy system.

Swapping in Pages

Swap-in takes place when a process attempts to address a page that has been swapped out to disk. The Page Fault exception handler triggers a swap-in operation when the following conditions occur (see the section "Handling a Faulty Address Inside the Address Space" in Chapter 9):

* The page including the address that caused the exception is a valid one—that is, it belongs to a memory region of the current process.
* The page is not present in memory—that is, the `Present` flag in the Page Table entry is cleared.
* The Page Table entry associated with the page is not null, but the `Dirty` bit is clear; this means that the entry contains a swapped-out page identifier (see the section "Demand Paging" in Chapter 9).

If all the above conditions are satisfied, `handle_pte_fault()` invokes a quite handy `do_swap_page()` function to swap in the page required.

The do_swap_page() function

The `do_swap_page()` function acts on the following parameters:

mm
    Memory descriptor address of the process that caused the Page Fault exception

vma
    Memory region descriptor address of the region that includes `address`

address
    Linear address that causes the exception

page_table
    Address of the Page Table entry that maps `address`

pmd
    Address of the Page Middle Directory that maps `address`

orig_pte
    Content of the Page Table entry that maps `address`

write_access
    Flag denoting whether the attempted access was a read or a write

Contrary to other functions, `do_swap_page()` never returns 0. It returns 1 if the page is already in the swap cache (minor fault), 2 if the page was read from the swap area (major fault), and −1 if an error occurred while performing the swap-in. It essentially executes the following steps:

1. Gets the swapped-out page identifier from `orig_pte`.

2. Invokes `pte_unmap()` to release any temporary kernel mapping for the Page Table created by the `handle_mm_fault()` function (see the section "Handling a Faulty Address Inside the Address Space" in Chapter 9). As explained in the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8, a kernel mapping is required to access a page table in high memory.

3. Releases the `page_table_lock` spin lock of the memory descriptor (it was acquired by the caller function `handle_pte_fault()`).

4. Invokes `lookup_swap_cache()` to check whether the swap cache already contains a page corresponding to the swapped-out page identifier; if the page is already in the swap cache, it jumps to step 6.

5. Invokes the `swapin_readahead()` function to read from the swap area a group of at most $2n$ pages, including the requested one. The value n is stored in the `page_cluster` variable, and is usually equal to 3.* Each page is read by invoking the `read_swap_cache_async()` function (see below).

6. Invokes `read_swap_cache_async()` once more to swap in precisely the page accessed by the process that caused the Page Fault. This step might appear redundant, but it isn't really. The `swapin_readahead()` function might fail in reading the requested page—for instance, because `page_cluster` is set to 0 or the function tried to read a group of pages including a free page slot or a defective page slot (`SWAP_MAP_BAD`). On the other hand, if `swapin_readahead()` succeeded, this invocation of `read_swap_cache_async()` terminates quickly because it finds the page in the swap cache.

7. If, despite all efforts, the requested page was not added to the swap cache, another kernel control path might have already swapped in the requested page on behalf of a clone of this process. This case is checked by temporarily acquiring the `page_table_lock` spin lock and comparing the entry to which `page_table` points with `orig_pte`. If they differ, the page has already been swapped in by some other kernel control path, so the function returns 1 (minor fault); otherwise, it returns −1 (failure).

8. At this point, we know that the page is in the swap cache. If the page has been effectively swapped in (major fault), the function invokes `grab_swap_token()` to try to grab the swap token (see the section "The Swap Token" earlier in this chapter).

9. Invokes `mark_page_accessed()` (see the earlier section "The Least Recently Used (LRU) Lists") and locks the page.

10. Acquires the `page_table_lock` spin lock.

---

* The system administrator may tune this value by writing into the */proc/sys/vm/page-cluster* file. Swap-in read-ahead can be disabled by setting `page_cluster` to 0.

11. Checks whether another kernel control path has swapped in the requested page on behalf of a clone of this process. In this case, it releases the `page_table_lock` spin lock, unlocks the page, and returns 1 (minor fault).

12. Invokes `swap_free( )` to decrease the usage counter of the page slot corresponding to `entry`.

13. Checks whether the swap cache is at least 50 percent full (`nr_swap_pages` is smaller than half of `total_swap_pages`). If so, it checks whether the page is owned only by the process that caused the fault (or one of its clones); if this is the case, removes the page from the swap cache.

14. Increases the `rss` field of the process's memory descriptor.

15. Updates the Page Table entry so the process can find the page. The function accomplishes this by writing the physical address of the requested page and the protection bits found in the `vm_page_prot` field of the memory region into the Page Table entry addressed by `page_table`. Moreover, if the access that caused the fault was a write and the faulting process is the unique owner of the page, the function also sets the `Dirty` flag and the `Read/Write` flag to prevent a useless Copy On Write fault.

16. Unlocks the page.

17. Invokes `page_add_anon_rmap( )` to insert the anonymous page in the object-based reverse mapping data structures (see the section "Reverse Mapping for Anonymous Pages" earlier in this chapter.)

18. If the `write_access` parameter is equal to 1, the function invokes `do_wp_page( )` to make a copy of the page frame (see the section "Copy On Write" in Chapter 9).

19. Releases the `mm->page_table_lock` spin lock and returns the `ret` return code: 1 (minor fault) or 2 (major fault).

The read_swap_cache_async( ) function

The `read_swap_cache_async( )` function is invoked whenever the kernel must swap in a page. It acts on three parameters:

entry
> A swapped-out page identifier

vma
> A pointer to the memory region that should contain the page

addr
> The linear address of the page

As we know, before accessing the swap partition, the function must check whether the swap cache already includes the desired page frame. Therefore, the function essentially executes the following operations:

1. Invokes `radix_tree_lookup( )` to locate in the radix tree of the `swapper_space` object a page frame at the position given by the swapped-out page identifier `entry`. If the page is found, it increases its reference counter and returns the address of its descriptor.

2. The page is not included in the swap cache. Invokes `alloc_pages( )` to allocate a new page frame. If no free page frame is available, it returns 0 (indicating the system is out of memory).

3. Invokes `add_to_swap_cache( )` to insert the page descriptor of the new page frame into the swap cache. As mentioned in the earlier section "Swap cache helper functions," this function also locks the page.

4. The previous step might fail if `add_to_swap_cache( )` finds a duplicate of the page in the swap cache. For instance, the process could block in step 2, thus allowing another process to start a swap-in operation on the same page slot. In this case, it releases the page frame allocated in step 2 and restarts from step 1.

5. Invokes `lru_cache_add_active()` to insert the page in the LRU active list (see the section "The Least Recently Used (LRU) Lists" earlier in this chapter).

6. The page descriptor of the new page frame is now in the swap cache. Invokes `swap_readpage( )` to read the page's contents from the swap area. This function is quite similar to `swap_writepage( )` described in the earlier section "Swapping Out Pages:" it clears the `PG_uptodate` flag of the page descriptor, invokes `get_swap_bio( )` to allocate and initialize a `bio` descriptor for the I/O transfer, and invokes `submit_bio( )` to submit the I/O request to the block subsystem layer.

7. Returns the address of the page descriptor.

The Ext2 and Ext3 Filesystems

In this chapter, we finish our extensive discussion of I/O and filesystems by taking a look at the details the kernel has to take care of when interacting with a specific filesystem. Because the Second Extended Filesystem (Ext2) is native to Linux and is used on virtually every Linux system, it is a natural choice for this discussion. Furthermore, Ext2 illustrates a lot of good practices in its support for modern filesystem features with fast performance. To be sure, other filesystems supported by Linux include many interesting features, but we have no room to examine all of them.

After introducing Ext2 in the section "General Characteristics of Ext2," we describe the data structures needed, just as in other chapters. Because we are looking at a specific way to store data on disk, we have to consider two versions of the same data structures. The section "Ext2 Disk Data Structures" shows the data structures stored by Ext2 on disk, while "Ext2 Memory Data Structures" shows the corresponding versions in memory.

Then we get to the operations performed on the filesystem. In the section "Creating the Ext2 Filesystem," we discuss how Ext2 is created in a disk partition. The next sections describe the kernel activities performed whenever the disk is used. Most of these are relatively low-level activities dealing with the allocation of disk space to inodes and data blocks.

In the last section, we give a short description of the Ext3 filesystem, which is the next step in the evolution of the Ext2 filesystem.

General Characteristics of Ext2

Unix-like operating systems use several types of filesystems. Although the files of all such filesystems have a common subset of attributes required by a few POSIX APIs such as stat(), each filesystem is implemented in a different way.

The first versions of Linux were based on the MINIX filesystem. As Linux matured, the *Extended Filesystem (Ext FS)* was introduced; it included several significant

extensions, but offered unsatisfactory performance. The *Second Extended Filesystem (Ext2)* was introduced in 1994; besides including several new features, it is quite efficient and robust and is, together with its offspring Ext3, the most widely used Linux filesystem.

The following features contribute to the efficiency of Ext2:

- When creating an Ext2 filesystem, the system administrator may choose the optimal block size (from 1,024 to 4,096 bytes), depending on the expected average file length. For instance, a 1,024-block size is preferable when the average file length is smaller than a few thousand bytes because this leads to less internal fragmentation—that is, less of a mismatch between the file length and the portion of the disk that stores it (see the section "Memory Area Management" in Chapter 8, where internal fragmentation for dynamic memory was discussed). On the other hand, larger block sizes are usually preferable for files greater than a few thousand bytes because this leads to fewer disk transfers, thus reducing system overhead.

- When creating an Ext2 filesystem, the system administrator may choose how many inodes to allow for a partition of a given size, depending on the expected number of files to be stored on it. This maximizes the effectively usable disk space.

- The filesystem partitions disk blocks into groups. Each group includes data blocks and inodes stored in adjacent tracks. Thanks to this structure, files stored in a single block group can be accessed with a lower average disk seek time.

- The filesystem *preallocates* disk data blocks to regular files before they are actually used. Thus, when the file increases in size, several blocks are already reserved at physically adjacent positions, reducing file fragmentation.

- Fast symbolic links (see the section "Hard and Soft Links" in Chapter 1) are supported. If the symbolic link represents a short pathname (at most 60 characters), it can be stored in the inode and can thus be translated without reading a data block.

Moreover, the Second Extended Filesystem includes other features that make it both robust and flexible:

- A careful implementation of file-updating that minimizes the impact of system crashes. For instance, when creating a new hard link for a file, the counter of hard links in the disk inode is increased first, and the new name is added into the proper directory next. In this way, if a hardware failure occurs after the inode update but before the directory can be changed, the directory is consistent, even if the inode's hard link counter is wrong. Deleting the file does not lead to catastrophic results, although the file's data blocks cannot be automatically reclaimed. If the reverse were done (changing the directory before updating the inode), the same hardware failure would produce a dangerous inconsistency: deleting the original hard link would remove its data blocks from disk, yet the new directory entry would refer to an inode that no longer exists. If that inode

number were used later for another file, writing into the stale directory entry would corrupt the new file.

- Support for automatic consistency checks on the filesystem status at boot time. The checks are performed by the *e2fsck* external program, which may be activated not only after a system crash, but also after a predefined number of filesystem mounts (a counter is increased after each mount operation) or after a predefined amount of time has elapsed since the most recent check.

- Support for *immutable* files (they cannot be modified, deleted, or renamed) and for *append-only* files (data can be added only to the end of them).

- Compatibility with both the Unix System V Release 4 and the BSD semantics of the user group ID for a new file. In SVR4, the new file assumes the user group ID of the process that creates it; in BSD, the new file inherits the user group ID of the directory containing it. Ext2 includes a mount option that specifies which semantic to use.

Even if the Ext2 filesystem is a mature, stable program, several additional features have been considered for inclusion. Some of them have already been coded and are available as external patches. Others are just planned, but in some cases, fields have already been introduced in the Ext2 inode for them. The most significant features being considered are:

Block fragmentation
System administrators usually choose large block sizes for accessing disks, because computer applications often deal with large files. As a result, small files stored in large blocks waste a lot of disk space. This problem can be solved by allowing several files to be stored in different fragments of the same block.

Handling of transparently compressed and encrypted files
These new options, which must be specified when creating a file, allow users to transparently store compressed and/or encrypted versions of their files on disk.

Logical deletion
An *undelete* option allows users to easily recover, if needed, the contents of a previously removed file.

Journaling
Journaling avoids the time-consuming check that is automatically performed on a filesystem when it is abruptly unmounted—for instance, as a consequence of a system crash.

In practice, none of these features has been officially included in the Ext2 filesystem. One might say that Ext2 is victim of its success; it has been the preferred filesystem adopted by most Linux distribution companies until a few years ago, and the millions of users who relied on it every day would have looked suspiciously at any attempt to replace Ext2 with some other filesystem.

The most compelling feature missing from Ext2 is journaling, which is required by high-availability servers. To provide for a smooth transition, journaling has not been introduced in the Ext2 filesystem; rather, as we'll discuss in the later section "The Ext3 Filesystem," a more recent filesystem that is fully compatible with Ext2 has been created, which also offers journaling. Users who do not really require journaling may continue to use the good old Ext2 filesystem, while the others will likely adopt the new filesystem. Nowadays, most distributions adopt Ext3 as the standard filesystem.

Ext2 Disk Data Structures

The first block in each Ext2 partition is never managed by the Ext2 filesystem, because it is reserved for the partition boot sector (see Appendix A). The rest of the Ext2 partition is split into *block groups*, each of which has the layout shown in Figure 18-1. As you will notice from the figure, some data structures must fit in exactly one block, while others may require more than one block. All the block groups in the filesystem have the same size and are stored sequentially, thus the kernel can derive the location of a block group in a disk simply from its integer index.
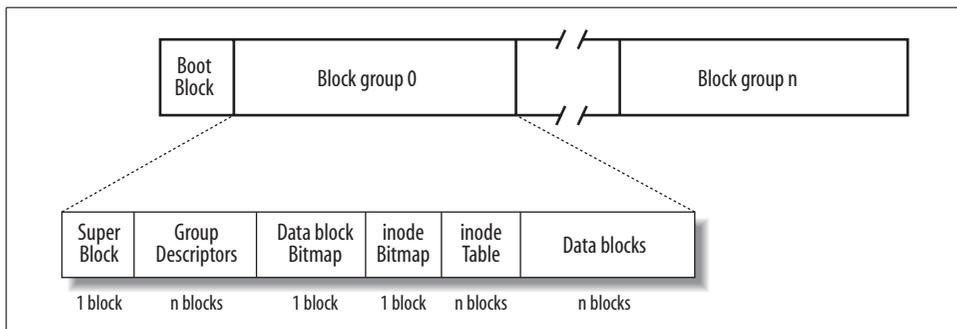
Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group

Block groups reduce file fragmentation, because the kernel tries to keep the data blocks belonging to a file in the same block group, if possible. Each block in a block group contains one of the following pieces of information:

- A copy of the filesystem's superblock
- A copy of the group of block group descriptors
- A data block bitmap
- An inode bitmap
- A table of inodes
- A chunk of data that belongs to a file; i.e., data blocks

If a block does not contain any meaningful information, it is said to be free.

As you can see from Figure 18-1, both the superblock and the group descriptors are duplicated in each block group. Only the superblock and the group descriptors included in block group 0 are used by the kernel, while the remaining superblocks and group descriptors are left unchanged; in fact, the kernel doesn't even look at them. When the *e2fsck* program executes a consistency check on the filesystem status, it refers to the superblock and the group descriptors stored in block group 0, and then copies them into all other block groups. If data corruption occurs and the main superblock or the main group descriptors in block group 0 become invalid, the system administrator can instruct *e2fsck* to refer to the old copies of the superblock and the group descriptors stored in a block groups other than the first. Usually, the redundant copies store enough information to allow *e2fsck* to bring the Ext2 partition back to a consistent state.

How many block groups are there? Well, that depends both on the partition size and the block size. The main constraint is that the block bitmap, which is used to identify the blocks that are used and free inside a group, must be stored in a single block. Therefore, in each block group, there can be at most 8×*b* blocks, where *b* is the block size in bytes. Thus, the total number of block groups is roughly $s/(8 \times b)$, where s is the partition size in blocks.

For example, let's consider a 32-GB Ext2 partition with a 4-KB block size. In this case, each 4-KB block bitmap describes 32K data blocks—that is, 128 MB. Therefore, at most 256 block groups are needed. Clearly, the smaller the block size, the larger the number of block groups.

Superblock

An Ext2 disk superblock is stored in an `ext2_super_block` structure, whose fields are listed in Table 18-1.[*] The `__u8`, `__u16`, and `__u32` data types denote unsigned numbers of length 8, 16, and 32 bits respectively, while the `__s8`, `__s16`, `__s32` data types denote signed numbers of length 8, 16, and 32 bits. To explicitly specify the order in which the bytes of a word or double-word are stored on disk, the kernel also makes use of the `__le16`, `__le32`, `__be16`, and `__be32` data types; the former two types denote the *little-endian ordering* for words and double-words (the least significant byte is stored at the highest address), respectively, while the latter two types denote the *big-endian ordering* (the most significant byte is stored at the highest address).

Table 18-1. The fields of the Ext2 superblock

Type	Field	Description
`__le32`	`s_inodes_count`	Total number of inodes
`__le32`	`s_blocks_count`	Filesystem size in blocks

---

[*] To ensure compatibility between the Ext2 and Ext3 filesystems, the `ext2_super_block` data structure includes some Ext3-specific fields, which are not shown in Table 18-1.

Table 18-1. The fields of the Ext2 superblock (continued)

Type	Field	Description
__le32	s_r_blocks_count	Number of reserved blocks
__le32	s_free_blocks_count	Free blocks counter
__le32	s_free_inodes_count	Free inodes counter
__le32	s_first_data_block	Number of first useful block (always 1)
__le32	s_log_block_size	Block size
__le32	s_log_frag_size	Fragment size
__le32	s_blocks_per_group	Number of blocks per group
__le32	s_frags_per_group	Number of fragments per group
__le32	s_inodes_per_group	Number of inodes per group
__le32	s_mtime	Time of last mount operation
__le32	s_wtime	Time of last write operation
__le16	s_mnt_count	Mount operations counter
__le16	s_max_mnt_count	Number of mount operations before check
__le16	s_magic	Magic signature
__le16	s_state	Status flag
__le16	s_errors	Behavior when detecting errors
__le16	s_minor_rev_level	Minor revision level
__le32	s_lastcheck	Time of last check
__le32	s_checkinterval	Time between checks
__le32	s_creator_os	OS where filesystem was created
__le32	s_rev_level	Revision level of the filesystem
__le16	s_def_resuid	Default UID for reserved blocks
__le16	s_def_resgid	Default user group ID for reserved blocks
__le32	s_first_ino	Number of first nonreserved inode
__le16	s_inode_size	Size of on-disk inode structure
__le16	s_block_group_nr	Block group number of this superblock
__le32	s_feature_compat	Compatible features bitmap
__le32	s_feature_incompat	Incompatible features bitmap
__le32	s_feature_ro_compat	Read-only compatible features bitmap
__u8 [16]	s_uuid	128-bit filesystem identifier
char [16]	s_volume_name	Volume name
char [64]	s_last_mounted	Pathname of last mount point
__le32	s_algorithm_usage_bitmap	Used for compression
__u8	s_prealloc_blocks	Number of blocks to preallocate
__u8	s_prealloc_dir_blocks	Number of blocks to preallocate for directories
__u16	s_padding1	Alignment to word
__u32 [204]	s_reserved	Nulls to pad out 1,024 bytes

The s_inodes_count field stores the number of inodes, while the s_blocks_count field stores the number of blocks in the Ext2 filesystem.

The s_log_block_size field expresses the block size as a power of 2, using 1,024 bytes as the unit. Thus, 0 denotes 1,024-byte blocks, 1 denotes 2,048-byte blocks, and so on. The s_log_frag_size field is currently equal to s_log_block_size, because block fragmentation is not yet implemented.

The s_blocks_per_group, s_frags_per_group, and s_inodes_per_group fields store the number of blocks, fragments, and inodes in each block group, respectively.

Some disk blocks are reserved to the superuser (or to some other user or group of users selected by the s_def_resuid and s_def_resgid fields). These blocks allow the system administrator to continue to use the filesystem even when no more free blocks are available for normal users.

The s_mnt_count, s_max_mnt_count, s_lastcheck, and s_checkinterval fields set up the Ext2 filesystem to be checked automatically at boot time. These fields cause *e2fsck* to run after a predefined number of mount operations has been performed, or when a predefined amount of time has elapsed since the last consistency check. (Both kinds of checks can be used together.) The consistency check is also enforced at boot time if the filesystem has not been cleanly unmounted (for instance, after a system crash) or when the kernel discovers some errors in it. The s_state field stores the value 0 if the filesystem is mounted or was not cleanly unmounted, 1 if it was cleanly unmounted, and 2 if it contains errors.

Group Descriptor and Bitmap

Each block group has its own group descriptor, an ext2_group_desc structure whose fields are illustrated in Table 18-2.

Table 18-2. The fields of the Ext2 group descriptor

Type	Field	Description
__le32	bg_block_bitmap	Block number of block bitmap
__le32	bg_inode_bitmap	Block number of inode bitmap
__le32	bg_inode_table	Block number of first inode table block
__le16	bg_free_blocks_count	Number of free blocks in the group
__le16	bg_free_inodes_count	Number of free inodes in the group
__le16	bg_used_dirs_count	Number of directories in the group
__le16	bg_pad	Alignment to word
__le32 [3]	bg_reserved	Nulls to pad out 24 bytes

The bg_free_blocks_count, bg_free_inodes_count, and bg_used_dirs_count fields are used when allocating new inodes and data blocks. These fields determine the most suitable block in which to allocate each data structure. The bitmaps are sequences of

bits, where the value 0 specifies that the corresponding inode or data block is free and the value 1 specifies that it is used. Because each bitmap must be stored inside a single block and because the block size can be 1,024, 2,048, or 4,096 bytes, a single bitmap describes the state of 8,192, 16,384, or 32,768 blocks.

Inode Table

The inode table consists of a series of consecutive blocks, each of which contains a predefined number of inodes. The block number of the first block of the inode table is stored in the bg_inode_table field of the group descriptor.

All inodes have the same size: 128 bytes. A 1,024-byte block contains 8 inodes, while a 4,096-byte block contains 32 inodes. To figure out how many blocks are occupied by the inode table, divide the total number of inodes in a group (stored in the s_inodes_per_group field of the superblock) by the number of inodes per block.

Each Ext2 inode is an ext2_inode structure whose fields are illustrated in Table 18-3.

Table 18-3. The fields of an Ext2 disk inode

Type	Field	Description
__le16	i_mode	File type and access rights
__le16	i_uid	Owner identifier
__le32	i_size	File length in bytes
__le32	i_atime	Time of last file access
__le32	i_ctime	Time that inode last changed
__le32	i_mtime	Time that file contents last changed
__le32	i_dtime	Time of file deletion
__le16	i_gid	User group identifier
__le16	i_links_count	Hard links counter
__le32	i_blocks	Number of data blocks of the file
__le32	i_flags	File flags
union	osd1	Specific operating system information
__le32 [EXT2_N_BLOCKS]	i_block	Pointers to data blocks
__le32	i_generation	File version (used when the file is accessed by a network filesystem)
__le32	i_file_acl	File access control list
__le32	i_dir_acl	Directory access control list
__le32	i_faddr	Fragment address
union	osd2	Specific operating system information

Many fields related to POSIX specifications are similar to the corresponding fields of the VFS's inode object and have already been discussed in the section "Inode

Objects" in Chapter 12. The remaining ones refer to the Ext2-specific implementation and deal mostly with block allocation.

In particular, the `i_size` field stores the effective length of the file in bytes, while the `i_blocks` field stores the number of data blocks (in units of 512 bytes) that have been allocated to the file.

The values of `i_size` and `i_blocks` are not necessarily related. Because a file is always stored in an integer number of blocks, a nonempty file receives at least one data block (since fragmentation is not yet implemented) and `i_size` may be smaller than 512×`i_blocks`. On the other hand, as we'll see in the section "File Holes" later in this chapter, a file may contain holes. In that case, `i_size` may be greater than 512×`i_blocks`.

The `i_block` field is an array of `EXT2_N_BLOCKS` (usually 15) pointers to blocks used to identify the data blocks allocated to the file (see the section "Data Blocks Addressing" later in this chapter).

The 32 bits reserved for the `i_size` field limit the file size to 4 GB. Actually, the highest-order bit of the `i_size` field is not used, so the maximum file size is limited to 2 GB. However, the Ext2 filesystem includes a "dirty trick" that allows larger files on systems that sport a 64-bit processor such as AMD's Opteron or IBM's PowerPC G5. Essentially, the `i_dir_acl` field of the inode, which is not used for regular files, represents a 32-bit extension of the `i_size` field. Therefore, the file size is stored in the inode as a 64-bit integer. The 64-bit version of the Ext2 filesystem is somewhat compatible with the 32-bit version because an Ext2 filesystem created on a 64-bit architecture may be mounted on a 32-bit architecture, and vice versa. On a 32-bit architecture, a large file cannot be accessed, unless opening the file with the `O_LARGEFILE` flag set (see the section "The open( ) System Call" in Chapter 12).

Recall that the VFS model requires each file to have a different inode number. In Ext2, there is no need to store on disk a mapping between an inode number and the corresponding block number because the latter value can be derived from the block group number and the relative position inside the inode table. For example, suppose that each block group contains 4,096 inodes and that we want to know the address on disk of inode 13,021. In this case, the inode belongs to the third block group and its disk address is stored in the 733rd entry of the corresponding inode table. As you can see, the inode number is just a key used by the Ext2 routines to retrieve the proper inode descriptor on disk quickly.

Extended Attributes of an Inode

The Ext2 inode format is a kind of straitjacket for filesystem designers. The length of an inode must be a power of 2 to avoid internal fragmentation in the blocks that store the inode table. Actually, most of the 128 characters of an Ext2 inode are currently packed with information, and there is little room left for additional fields. On the other hand, expanding the inode length to 256 would be quite wasteful, besides

introducing compatibility problems between Ext2 filesystems that use different inode lengths.

Extended attributes have been introduced to overcome the above limitation. These attributes are stored on a disk block allocated outside of any inode. The `i_file_acl` field of an inode points to the block containing the extended attributes. Different inodes that have the same set of extended attributes may share the same block.

Each extended attribute has a name and a value. Both of them are encoded as variable length arrays of characters, as specified by the `ext2_xattr_entry` descriptor. Figure 18-2 shows the layout in Ext2 of the extended attributes inside a block. Each attribute is split in two parts: the `ext2_xattr_entry` descriptor together with the name of the attribute are placed at the beginning of the block, while the value of the attribute is placed at the end of the block. The entries at the beginning of the block are ordered according to the attribute names, while the positions of the values are fixed, because they are determined by the allocation order of the attributes.
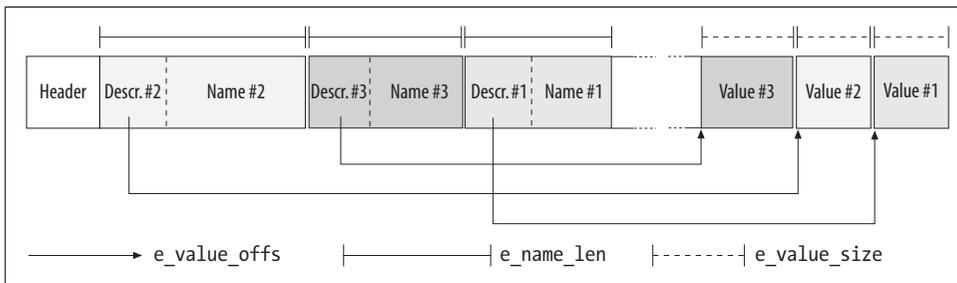
Figure 18-2. Layout of a block containing extended attributes

There are many system calls used to set, retrieve, list, and remove the extended attributes of a file. The `setxattr()`, `lsetxattr()`, and `fsetxattr()` system calls set an extended attribute of a file; essentially, they differ in how symbolic links are handled, and in how the file is specified (either passing a pathname or a file descriptor). Similarly, the `getxattr()`, `lgetxattr()`, and `fgetxattr()` system calls return the value of an extended attribute. The `listxattr()`, `llistxattr()`, and `flistxattr()` list all extended attributes of a file. Finally, the `removexattr()`, `lremovexattr()`, and `fremovexattr()` system calls remove an extended attribute from a file.

Access Control Lists

Access control lists were proposed a long time ago to improve the file protection mechanism in Unix filesystems. Instead of classifying the users of a file under three classes—owner, group, and others—an *access control list* (*ACL*) can be associated with each file. Thanks to this kind of list, a user may specify for each of his files the names of specific users (or groups of users) and the privileges to be given to these users.

Linux 2.6 fully supports ACLs by making use of inode extended attributes. As a matter of fact, extended attributes have been introduced mainly to support ACLs. Therefore, the chacl(), setfacl(), and getfacl() library functions, which allow you to manipulate the ACLs of a file, rely essentially upon the setxattr() and getxattr() system calls introduced in the previous section.

Unfortunately, the outcome of a working group that defined security extensions within the POSIX 1003.1 family of standards has never been formalized as a new POSIX standard. As a result, ACLs are supported nowadays on different filesystem types on many UNIX-like systems, albeit with a number of subtle differences among the different implementations.

How Various File Types Use Disk Blocks

The different types of files recognized by Ext2 (regular files, pipes, etc.) use data blocks in different ways. Some files store no data and therefore need no data blocks at all. This section discusses the storage requirements for each type, which are listed in Table 18-4.

Table 18-4. Ext2 file types

File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Regular file

Regular files are the most common case and receive almost all the attention in this chapter. But a regular file needs data blocks only when it starts to have data. When first created, a regular file is empty and needs no data blocks; it can also be emptied by the truncate() or open() system calls. Both situations are common; for instance, when you issue a shell command that includes the string >*filename*, the shell creates an empty file or truncates an existing one.

Directory

Ext2 implements directories as a special kind of file whose data blocks store filenames together with the corresponding inode numbers. In particular, such data blocks contain structures of type ext2_dir_entry_2. The fields of that structure are

shown in Table 18-5. The structure has a variable length, because the last name field is a variable length array of up to EXT2_NAME_LEN characters (usually 255). Moreover, for reasons of efficiency, the length of a directory entry is always a multiple of 4 and, therefore, null characters (\0) are added for padding at the end of the filename, if necessary. The name_len field stores the actual filename length (see Figure 18-3).

Table 18-5. The fields of an Ext2 directory entry

Type	Field	Description
__le32	inode	Inode number
__le16	rec_len	Directory entry length
__u8	name_len	Filename length
__u8	file_type	File type
char [EXT2_NAME_LEN]	name	Filename

The file_type field stores a value that specifies the file type (see Table 18-4). The rec_len field may be interpreted as a pointer to the next valid directory entry: it is the offset to be added to the starting address of the directory entry to get the starting address of the next valid directory entry. To delete a directory entry, it is sufficient to set its inode field to 0 and suitably increment the value of the rec_len field of the previous valid entry. Read the rec_len field of Figure 18-3 carefully; you'll see that the *oldfile* entry was deleted because the rec_len field of *usr* is set to 12+16 (the lengths of the *usr* and *oldfile* entries).
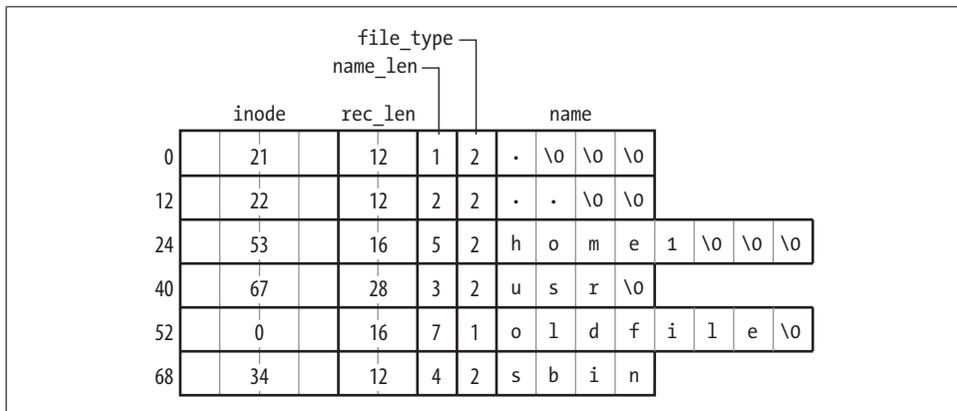
Figure 18-3. An example of the Ext2 directory

Symbolic link

As stated before, if the pathname of a symbolic link has up to 60 characters, it is stored in the i_block field of the inode, which consists of an array of 15 4-byte integers; no data block is therefore required. If the pathname is longer than 60 characters, however, a single data block is required.

Device file, pipe, and socket

No data blocks are required for these kinds of files. All the necessary information is stored in the inode.

Ext2 Memory Data Structures

For the sake of efficiency, most information stored in the disk data structures of an Ext2 partition are copied into RAM when the filesystem is mounted, thus allowing the kernel to avoid many subsequent disk read operations. To get an idea of how often some data structures change, consider some fundamental operations:

- When a new file is created, the values of the s_free_inodes_count field in the Ext2 superblock and of the bg_free_inodes_count field in the proper group descriptor must be decreased.

- If the kernel appends some data to an existing file so that the number of data blocks allocated for it increases, the values of the s_free_blocks_count field in the Ext2 superblock and of the bg_free_blocks_count field in the group descriptor must be modified.

- Even just rewriting a portion of an existing file involves an update of the s_wtime field of the Ext2 superblock.

Because all Ext2 disk data structures are stored in blocks of the Ext2 partition, the kernel uses the page cache to keep them up-to-date (see the section "Writing Dirty Pages to Disk" in Chapter 15).

Table 18-6 specifies, for each type of data related to Ext2 filesystems and files, the data structure used on the disk to represent its data, the data structure used by the kernel in memory, and a rule of thumb used to determine how much caching is used. Data that is updated very frequently is always cached; that is, the data is permanently stored in memory and included in the page cache until the corresponding Ext2 partition is unmounted. The kernel gets this result by keeping the page's usage counter greater than 0 at all times.

Table 18-6. VFS images of Ext2 data structures

Type	Disk data structure	Memory data structure	Caching mode
Superblock	ext2_super_block	ext2_sb_info	Always cached
Group descriptor	ext2_group_desc	ext2_group_desc	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	ext2_inode	ext2_inode_info	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	ext2_inode	None	Never
Free block	Array of bytes	None	Never

The never-cached data is not kept in any cache because it does not represent meaningful information. Conversely, the always-cached data is always present in RAM, thus it is never necessary to read the data from disk (periodically, however, the data must be written back to disk). In between these extremes lies the *dynamic* mode. In this mode, the data is kept in a cache as long as the associated object (inode, data block, or bitmap) is in use; when the file is closed or the data block is deleted, the page frame reclaiming algorithm may remove the associated data from the cache.

It is interesting to observe that inode and block bitmaps are not kept permanently in memory; rather, they are read from disk when needed. Actually, many disk reads are avoided thanks to the page cache, which keeps in memory the most recently used disk blocks (see the section "Storing Blocks in the Page Cache" in Chapter 15).[*]

The Ext2 Superblock Object

As stated in the section "Superblock Objects" in Chapter 12, the `s_fs_info` field of the VFS superblock points to a structure containing filesystem-specific data. In the case of Ext2, this field points to a structure of type `ext2_sb_info`, which includes the following information:

- Most of the disk superblock fields
- An `s_sbh` pointer to the buffer head of the buffer containing the disk superblock
- An `s_es` pointer to the buffer containing the disk superblock
- The number of group descriptors, `s_desc_per_block`, that can be packed in a block
- An `s_group_desc` pointer to an array of buffer heads of buffers containing the group descriptors (usually, a single entry is sufficient)
- Other data related to mount state, mount options, and so on

Figure 18-4 shows the links between the `ext2_sb_info` data structures and the buffers and buffer heads relative to the Ext2 superblock and to the group descriptors.

When the kernel mounts an Ext2 filesystem, it invokes the `ext2_fill_super()` function to allocate space for the data structures and to fill them with data read from disk (see the section "Mounting a Generic Filesystem" in Chapter 12). This is a simplified description of the function, which emphasizes the memory allocations for buffers and descriptors:

1. Allocates an `ext2_sb_info` descriptor and stores its address in the `s_fs_info` field of the superblock object passed as the parameter.

---

[*] In Linux 2.4 and earlier versions, the most recently used inode and block bitmaps were stored in ad-hoc caches of bounded size.
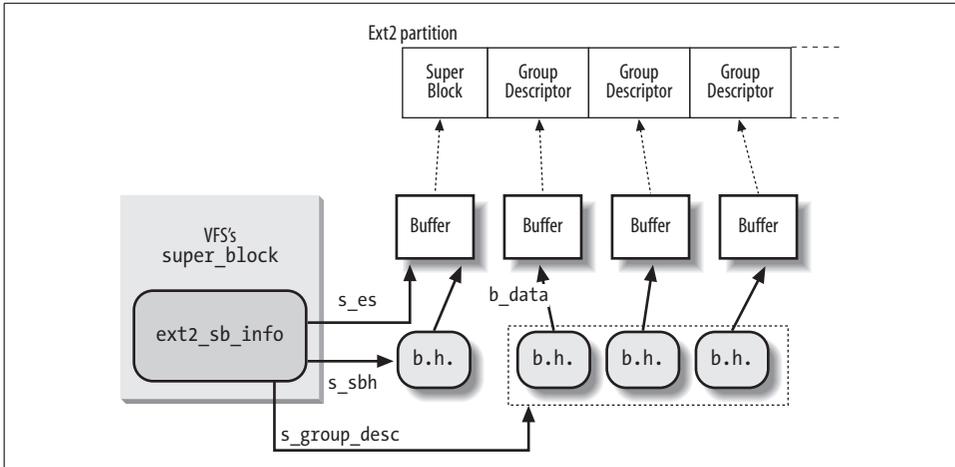
Figure 18-4. The ext2_sb_info data structure

2. Invokes _ _bread() to allocate a buffer in a buffer page together with the corresponding buffer head, and to read the superblock from disk into the buffer; as discussed in the section "Searching Blocks in the Page Cache" in Chapter 15, no allocation is performed if the block is already stored in a buffer page in the page cache and it is up-to-date. Stores the buffer head address in the s_sbh field of the Ext2 superblock object.

3. Allocates an array of bytes—one byte for each group—and stores its address in the s_debts field of the ext2_sb_info descriptor (see the section "Creating inodes" later in this chapter).

4. Allocates an array of pointers to buffer heads, one for each group descriptor, and stores the address of the array in the s_group_desc field of the ext2_sb_info descriptor.

5. Invokes repeatedly _ _bread() to allocate buffers and to read from disk the blocks containing the Ext2 group descriptors; stores the addresses of the buffer heads in the s_group_desc array allocated in the previous step.

6. Allocates an inode and a dentry object for the root directory, and sets up a few fields of the superblock object so that it will be possible to read the root inode from disk.

Clearly, all the data structures allocated by ext2_fill_super() are kept in memory after the function returns; they will be released only when the Ext2 filesystem will be unmounted. When the kernel must modify a field in the Ext2 superblock, it simply writes the new value in the proper position of the corresponding buffer and then marks the buffer as dirty.

The Ext2 inode Object

When opening a file, a pathname lookup is performed. For each component of the pathname that is not already in the dentry cache, a new dentry object and a new inode object are created (see the section "Standard Pathname Lookup" in Chapter 12). When the VFS accesses an Ext2 disk inode, it creates a corresponding *inode descriptor* of type ext2_inode_info. This descriptor includes the following information:

- The whole VFS inode object (see Table 12-3 in Chapter 12) stored in the field vfs_inode

- Most of the fields found in the disk's inode structure that are not kept in the VFS inode

- The i_block_group block group index at which the inode belongs (see the section "Ext2 Disk Data Structures" earlier in this chapter)

- The i_next_alloc_block and i_next_alloc_goal fields, which store the logical block number and the physical block number of the disk block that was most recently allocated to the file, respectively

- The i_prealloc_block and i_prealloc_count fields, which are used for data block preallocation (see the section "Allocating a Data Block" later in this chapter)

- The xattr_sem field, a read/write semaphore that allows extended attributes to be read concurrently with the file data

- The i_acl and i_default_acl fields, which point to the ACLs of the file

When dealing with Ext2 files, the alloc_inode superblock method is implemented by means of the ext2_alloc_inode() function. It gets first an ext2_inode_info descriptor from the ext2_inode_cachep slab allocator cache, then it returns the address of the inode object embedded in the new ext2_inode_info descriptor.

Creating the Ext2 Filesystem

There are generally two stages to creating a filesystem on a disk. The first step is to format it so that the disk driver can read and write blocks on it. Modern hard disks come preformatted from the factory and need not be reformatted; floppy disks may be formatted on Linux using a utility program such as *superformat* or *fdformat*. The second step involves creating a filesystem, which means setting up the structures described in detail earlier in this chapter.

Ext2 filesystems are created by the *mke2fs* utility program; it assumes the following default options, which may be modified by the user with flags on the command line:

- Block size: 1,024 bytes (default value for a small filesystem)
- Fragment size: block size (block fragmentation is not implemented)
- Number of allocated inodes: 1 inode for each 8,192 bytes
- Percentage of reserved blocks: 5 percent

The program performs the following actions:

1. Initializes the superblock and the group descriptors.
2. Optionally, checks whether the partition contains defective blocks; if so, it creates a list of defective blocks.
3. For each block group, reserves all the disk blocks needed to store the superblock, the group descriptors, the inode table, and the two bitmaps.
4. Initializes the inode bitmap and the data map bitmap of each block group to 0.
5. Initializes the inode table of each block group.
6. Creates the */root* directory.
7. Creates the *lost+found* directory, which is used by *e2fsck* to link the lost and found defective blocks.
8. Updates the inode bitmap and the data block bitmap of the block group in which the two previous directories have been created.
9. Groups the defective blocks (if any) in the *lost+found* directory.

Let's consider how an Ext2 1.44 MB floppy disk is initialized by *mke2fs* with the default options.

Once mounted, it appears to the VFS as a volume consisting of 1,412 blocks; each one is 1,024 bytes in length. To examine the disk's contents, we can execute the Unix command:

```
$ dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

to get a file containing the hexadecimal dump of the floppy disk contents in the */tmp* directory.*

By looking at that file, we can see that, due to the limited capacity of the disk, a single group descriptor is sufficient. We also notice that the number of reserved blocks is set to 72 (5 percent of 1,440) and, according to the default option, the inode table must include 1 inode for each 8,192 bytes—that is, 184 inodes stored in 23 blocks.

Table 18-7 summarizes how the Ext2 filesystem is created on a floppy disk when the default options are selected.

Table 18-7. Ext2 block allocation for a floppy disk

Block	Content
0	Boot block
1	Superblock
2	Block containing a single block group descriptor
3	Data block bitmap

---

* Most information on an Ext2 filesystem could also be obtained by using the *dumpe2fs* and *debugfs* utility programs.

Table 18-7. Ext2 block allocation for a floppy disk (continued)

Block	Content
4	inode bitmap
5–27	inode table: inodes up to 10: reserved (inode 2 is the root); inode 11: *lost+found*; inodes 12–184: free
28	Root directory (includes ., .., and *lost+found*)
29	*lost+found* directory (includes . and ..)
30–40	Reserved blocks preallocated for *lost+found* directory
41–1439	Free blocks

Ext2 Methods

Many of the VFS methods described in Chapter 12 have a corresponding Ext2 implementation. Because it would take a whole book to describe all of them, we limit ourselves to briefly reviewing the methods implemented in Ext2. Once the disk and the memory data structures are clearly understood, the reader should be able to follow the code of the Ext2 functions that implement them.

Ext2 Superblock Operations

Many VFS superblock operations have a specific implementation in Ext2, namely alloc_inode, destroy_inode, read_inode, write_inode, delete_inode, put_super, write_super, statfs, remount_fs, and clear_inode. The addresses of the superblock methods are stored in the ext2_sops array of pointers.

Ext2 inode Operations

Some of the VFS inode operations have a specific implementation in Ext2, which depends on the type of the file to which the inode refers.

The inode operations for Ext2 regular files and Ext2 directories are shown in Table 18-8; the purpose of each method is described in the section "Inode Objects" in Chapter 12. The table does not show the methods that are undefined (a NULL pointer) for both regular files and directories; recall that if a method is undefined, the VFS either invokes a generic function or does nothing at all. The addresses of the Ext2 methods for regular files and directories are stored in the ext2_file_inode_operations and ext2_dir_inode_operations tables, respectively.

Table 18-8. Ext2 inode operations for regular files and directories

VFS inode operation	Regular file	Directory
create	NULL	ext2_create()
lookup	NULL	ext2_lookup()
link	NULL	ext2_link()
unlink	NULL	ext2_unlink()

VFS inode operation	Regular file	Directory
symlink	NULL	ext2_symlink()
mkdir	NULL	ext2_mkdir()
rmdir	NULL	ext2_rmdir()
mknod	NULL	ext2_mknod()
rename	NULL	ext2_rename()
truncate	ext2_truncate()	NULL
permission	ext2_permission()	ext2_permission()
setattr	ext2_setattr()	ext2_setattr()
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

The inode operations for Ext2 symbolic links are shown in Table 18-9 (undefined methods have been omitted). Actually, there are two types of symbolic links: the fast symbolic links represent pathnames that can be fully stored inside the inodes, while the regular symbolic links represent longer pathnames. Accordingly, there are two sets of inode operations, which are stored in the ext2_fast_symlink_inode_operations and ext2_symlink_inode_operations tables, respectively.

Table 18-9. Ext2 inode operations for fast and regular symbolic links

VFS inode operation	Fast symbolic link	Regular symbolic link
readlink	generic_readlink()	generic_readlink()
follow_link	ext2_follow_link()	page_follow_link_light()
put_link	NULL	page_put_link()
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

If the inode refers to a character device file, to a block device file, or to a named pipe (see "FIFOs" in Chapter 19), the inode operations do not depend on the filesystem. They are specified in the chrdev_inode_operations, blkdev_inode_operations, and fifo_inode_operations tables, respectively.

Ext2 File Operations

The file operations specific to the Ext2 filesystem are listed in Table 18-10. As you can see, several VFS methods are implemented by generic functions that are common

to many filesystems. The addresses of these methods are stored in the `ext2_file_operations` table.

Table 18-10. Ext2 file operations

VFS file operation	Ext2 method
llseek	generic_file_llseek()
read	generic_file_read()
write	generic_file_write()
aio_read	generic_file_aio_read()
aio_write	generic_file_aio_write()
ioctl	ext2_ioctl()
mmap	generic_file_mmap()
open	generic_file_open()
release	ext2_release_file()
fsync	ext2_sync_file()
readv	generic_file_readv()
writev	generic_file_writev()
sendfile	generic_file_sendfile()

Notice that the Ext2's read and write methods are implemented by the `generic_file_read()` and `generic_file_write()` functions, respectively. These are described in the sections "Reading from a File" and "Writing to a File" in Chapter 16.

Managing Ext2 Disk Space

The storage of a file on disk differs from the view the programmer has of the file in two ways: blocks can be scattered around the disk (although the filesystem tries hard to keep blocks sequential to improve access time), and files may appear to a programmer to be bigger than they really are because a program can introduce holes into them (through the `lseek()` system call).

In this section, we explain how the Ext2 filesystem manages the disk space—how it allocates and deallocates inodes and data blocks. Two main problems must be addressed:

- Space management must make every effort to avoid *file fragmentation*—the physical storage of a file in several, small pieces located in non-adjacent disk blocks. File fragmentation increases the average time of sequential read operations on the files, because the disk heads must be frequently repositioned during

the read operation.* This problem is similar to the external fragmentation of RAM discussed in the section "The Buddy System Algorithm" in Chapter 8.

- Space management must be time-efficient; that is, the kernel should be able to quickly derive from a file offset the corresponding logical block number in the Ext2 partition. In doing so, the kernel should limit as much as possible the number of accesses to addressing tables stored on disk, because each such intermediate access considerably increases the average file access time.

Creating inodes

The `ext2_new_inode( )` function creates an Ext2 disk inode, returning the address of the corresponding inode object (or `NULL`, in case of failure). The function carefully selects the block group that contains the new inode; this is done to spread unrelated directories among different groups and, at the same time, to put files into the same group as their parent directories. To balance the number of regular files and directories in a block group, Ext2 introduces a "debt" parameter for every block group.

The function acts on two parameters: the address `dir` of the inode object that refers to the directory into which the new inode must be inserted and a `mode` that indicates the type of inode being created. The latter argument also includes the `MS_SYNCHRONOUS` mount flag (see the section "Mounting a Generic Filesystem" in Chapter 12) that requires the current process to be suspended until the inode is allocated. The function performs the following actions:

1. Invokes `new_inode( )` to allocate a new VFS inode object; initializes its `i_sb` field to the superblock address stored in `dir->i_sb`, and adds it to the in-use inode list and to the superblock's list (see the section "Inode Objects" in Chapter 12).

2. If the new inode is a directory, the function invokes `find_group_orlov( )` to find a suitable block group for the directory.† This function implements the following heuristics:

   a. Directories having as parent the filesystem root should be spread among all block groups. Thus, the function searches the block groups looking for a group having a number of free inodes and a number of free blocks above the average. If there is no such group, it jumps to step 2c.

   b. Nested directories—not having the filesystem root as parent—should be put in the group of the parent if it satisfies the following rules:
      - The group does not contain too many directories
      - The group has a sufficient number of free inodes left

---

* Please note that fragmenting a file across block groups (A Bad Thing) is quite different from the not-yet-implemented fragmentation of blocks to store many files in one block (A Good Thing).

† The Ext2 filesystem may also be mounted with an option flag that forces the kernel to make use of a simpler, older allocation strategy, which is implemented by the `find_group_dir( )` function.

- The group has a small "debt" (the debt of a block group is stored in the array of counters pointed to by the s_debts field of the ext2_sb_info descriptor; the debt is increased each time a new directory is added and decreased each time another type of file is added)

   If the parent's group does not satisfy these rules, it picks the first group that satisfies them. If no such group exists, it jumps to step 2c.

   c. This is the "fallback" rule, to be used if no good group has been found. The function starts with the block group containing the parent directory and selects the first block group that has more free inodes than the average number of free inodes per block group.

3. If the new inode is not a directory, it invokes find_group_other( ) to allocate it in a block group having a free inode. This function selects the group by starting from the one that contains the parent directory and moving farther away from it; to be precise:

   a. Performs a quick logarithmic search starting from the block group that includes the parent directory dir. The algorithm searches log(n) block groups, where n is the total number of block groups. The algorithm jumps further ahead until it finds an available block group—for example, if we call the number of the starting block group i, the algorithm considers block groups $i \bmod (n)$, $i+1 \bmod (n)$, $i+1+2 \bmod (n)$, $i+1+2+4 \bmod (n)$, etc.

   b. If the logarithmic search failed in finding a block group with a free inode, the function performs an exhaustive linear search starting from the block group that includes the parent directory dir.

4. Invokes read_inode_bitmap( ) to get the inode bitmap of the selected block group and searches for the first null bit into it, thus obtaining the number of the first free disk inode.

5. Allocates the disk inode: sets the corresponding bit in the inode bitmap and marks the buffer containing the bitmap as dirty. Moreover, if the filesystem has been mounted specifying the MS_SYNCHRONOUS flag (see the section "Mounting a Generic Filesystem" in Chapter 12), the function invokes sync_dirty_buffer( ) to start the I/O write operation and waits until the operation terminates.

6. Decreases the bg_free_inodes_count field of the group descriptor. If the new inode is a directory, the function increases the bg_used_dirs_count field and marks the buffer containing the group descriptor as dirty.

7. Increases or decreases the group's counter in the s_debts array of the superblock, according to whether the inode refers to a regular file or a directory.

8. Decreases the s_freeinodes_counter field of the ext2_sb_info data structure; moreover, if the new inode is a directory, it increases the s_dirs_counter field in the ext2_sb_info data structure.

9. Sets the s_dirt flag of the superblock to 1, and marks the buffer that contains it to as dirty.

10. Sets the s_dirt field of the VFS's superblock object to 1.

11. Initializes the fields of the inode object. In particular, it sets the inode number i_no and copies the value of xtime.tv_sec into i_atime, i_mtime, and i_ctime. Also loads the i_block_group field in the ext2_inode_info structure with the block group index. Refer to Table 18-3 for the meaning of these fields.

12. Initializes the ACLs of the inode.

13. Inserts the new inode object into the hash table inode_hashtable and invokes mark_inode_dirty() to move the inode object into the superblock's dirty inode list (see the section "Inode Objects" in Chapter 12).

14. Invokes ext2_preread_inode() to read from disk the block containing the inode and to put the block in the page cache. This type of read-ahead is done because it is likely that a recently created inode will be written back soon.

15. Returns the address of the new inode object.

Deleting inodes

The ext2_free_inode() function deletes a disk inode, which is identified by an inode object whose address inode is passed as the parameter. The kernel should invoke the function after a series of cleanup operations involving internal data structures and the data in the file itself. It should come after the inode object has been removed from the inode hash table, after the last hard link referring to that inode has been deleted from the proper directory and after the file is truncated to 0 length to reclaim all its data blocks (see the section "Releasing a Data Block" later in this chapter). It performs the following actions:

1. Invokes clear_inode(), which in turn executes the following operations:

    a. Removes any dirty "indirect" buffer associated with the inode (see the later section "Data Blocks Addressing"); they are collected in the list headed at the private_list field of the address_space object inode->i_data (see the section "The address_space Object" in Chapter 15).

    b. If the I_LOCK flag of the inode is set, some of the inode's buffers are involved in I/O data transfers; the function suspends the current process until these I/O data transfers terminate.

    c. Invokes the clear_inode method of the superblock object, if defined; the Ext2 filesystem does not define it.

    d. If the inode refers to a device file, it removes the inode object from the device's list of inodes; this list is rooted either in the list field of the cdev character device descriptor (see the section "Character Device Drivers" in Chapter 13) or in the bd_inodes field of the block_device block device descriptor (see the section "Block Devices" in Chapter 14).

    e. Sets the state of the inode to I_CLEAR (the inode object contents are no longer meaningful).

2. Computes the index of the block group containing the disk inode from the inode number and the number of inodes in each block group.

3. Invokes `read_inode_bitmap( )` to get the inode bitmap.

4. Increases the `bg_free_inodes_count( )` field of the group descriptor. If the deleted inode is a directory, it decreases the `bg_used_dirs_count` field. Marks the buffer that contains the group descriptor as dirty.

5. If the deleted inode is a directory, it decreases the `s_dirs_counter` field in the `ext2_sb_info` data structure, sets the `s_dirt` flag of the superblock to 1, and marks the buffer that contains it as dirty.

6. Clears the bit corresponding to the disk inode in the inode bitmap and marks the buffer that contains the bitmap as dirty. Moreover, if the filesystem has been mounted with the `MS_SYNCHRONIZE` flag, it invokes `sync_dirty_buffer( )` to wait until the write operation on the bitmap's buffer terminates.

Data Blocks Addressing

Each nonempty regular file consists of a group of data blocks. Such blocks may be referred to either by their relative position inside the file —their file block number— or by their position inside the disk partition—their logical block number (see the section "Block Devices Handling" in Chapter 14).

Deriving the logical block number of the corresponding data block from an offset f inside a file is a two-step process:

1. Derive from the offset f the file block number—the index of the block that contains the character at offset f.

2. Translate the file block number to the corresponding logical block number.

Because Unix files do not include any control characters, it is quite easy to derive the file block number containing the fth character of a file: simply take the quotient of f and the filesystem's block size and round down to the nearest integer.

For instance, let's assume a block size of 4 KB. If f is smaller than 4,096, the character is contained in the first data block of the file, which has file block number 0. If f is equal to or greater than 4,096 and less than 8,192, the character is contained in the data block that has file block number 1, and so on.

This is fine as far as file block numbers are concerned. However, translating a file block number into the corresponding logical block number is not nearly as straightforward, because the data blocks of an Ext2 file are not necessarily adjacent on disk.

The Ext2 filesystem must therefore provide a method to store the connection between each file block number and the corresponding logical block number on disk. This mapping, which goes back to early versions of Unix from AT&T, is implemented partly inside the inode. It also involves some specialized blocks that contain extra pointers, which are an inode extension used to handle large files.

The i_block field in the disk inode is an array of EXT2_N_BLOCKS components that contain logical block numbers. In the following discussion, we assume that EXT2_N_BLOCKS has the default value, namely 15. The array represents the initial part of a larger data structure, which is illustrated in Figure 18-5. As can be seen in the figure, the 15 components of the array are of 4 different types:

- The first 12 components yield the logical block numbers corresponding to the first 12 blocks of the file—to the blocks that have file block numbers from 0 to 11.

- The component at index 12 contains the logical block number of a block, called *indirect block*, that represents a second-order array of logical block numbers. They correspond to the file block numbers ranging from 12 to $b/4+11$, where b is the filesystem's block size (each logical block number is stored in 4 bytes, so we divide by 4 in the formula). Therefore, the kernel must look in this component for a pointer to a block, and then look in that block for another pointer to the ultimate block that contains the file contents.

- The component at index 13 contains the logical block number of an indirect block containing a second-order array of logical block numbers; in turn, the entries of this second-order array point to third-order arrays, which store the logical block numbers that correspond to the file block numbers ranging from $b/4+12$ to $(b/4)^2+(b/4)+11$.

- Finally, the component at index 14 uses triple indirection: the fourth-order arrays store the logical block numbers corresponding to the file block numbers ranging from $(b/4)^2+(b/4)+12$ to $(b/4)^3+(b/4)^2+(b/4)+11$.
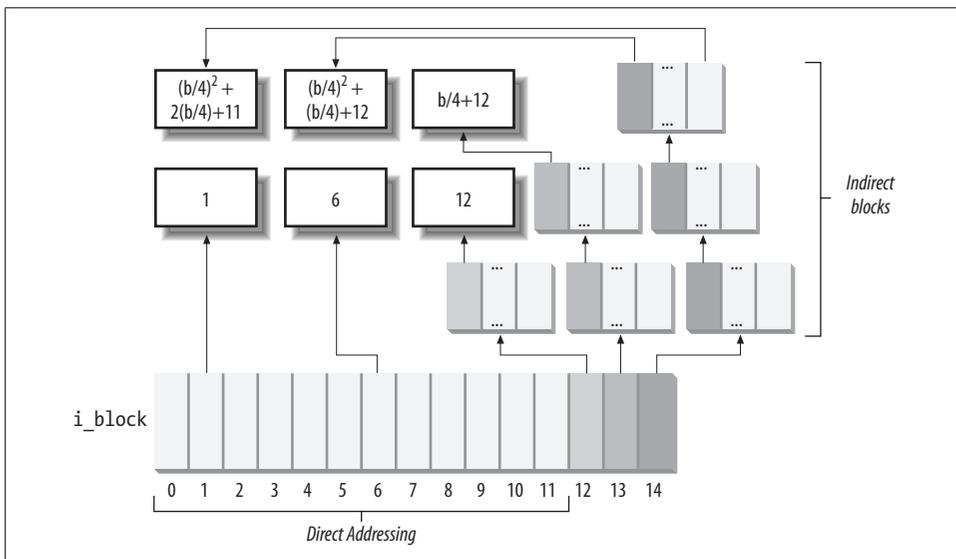
Figure 18-5. Data structures used to address the file's data blocks

In Figure 18-5, the number inside a block represents the corresponding file block number. The arrows, which represent logical block numbers stored in array components,

show how the kernel finds its way through indirect blocks to reach the block that contains the actual contents of the file.

Notice how this mechanism favors small files. If the file does not require more than 12 data blocks, every data can be retrieved in two disk accesses: one to read a component in the i_block array of the disk inode and the other to read the requested data block. For larger files, however, three or even four consecutive disk accesses may be needed to access the required block. In practice, this is a worst-case estimate, because dentry, inode, and page caches contribute significantly to reduce the number of real disk accesses.

Notice also how the block size of the filesystem affects the addressing mechanism, because a larger block size allows the Ext2 to store more logical block numbers inside a single block. Table 18-11 shows the upper limit placed on a file's size for each block size and each addressing mode. For instance, if the block size is 1,024 bytes and the file contains up to 268 kilobytes of data, the first 12 KB of a file can be accessed through direct mapping and the remaining 13–268 KB can be addressed through simple indirection. Files larger than 2 GB must be opened on 32-bit architectures by specifying the O_LARGEFILE opening flag.

Table 18-11. File-size upper limits for data block addressing

Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1,024	12 KB	268 KB	64.26 MB	16.06 GB
2,048	24 KB	1.02 MB	513.02 MB	256.5 GB
4,096	48 KB	4.04 MB	4 GB	~ 4 TB

File Holes

A *file hole* is a portion of a regular file that contains null characters and is not stored in any data block on disk. Holes are a long-standing feature of Unix files. For instance, the following Unix command creates a file in which the first bytes are a hole:

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

Now */tmp/hole* has 6,145 characters (6,144 null characters plus an X character), yet the file occupies just one data block on disk.

File holes were introduced to avoid wasting disk space. They are used extensively by database applications and, more generally, by all applications that perform hashing on files.

The Ext2 implementation of file holes is based on dynamic data block allocation: a block is actually assigned to a file only when the process needs to write data into it. The i_size field of each inode defines the size of the file as seen by the program, including the holes, while the i_blocks field stores the number of data blocks effectively assigned to the file (in units of 512 bytes).

In our earlier example of the dd command, suppose the */tmp/hole* file was created on an Ext2 partition that has blocks of size 4,096. The i_size field of the corresponding disk inode stores the number 6,145, while the i_blocks field stores the number 8 (because each 4,096-byte block includes eight 512-byte blocks). The second element of the i_block array (corresponding to the block having file block number 1) stores the logical block number of the allocated block, while all other elements in the array are null (see Figure 18-6).
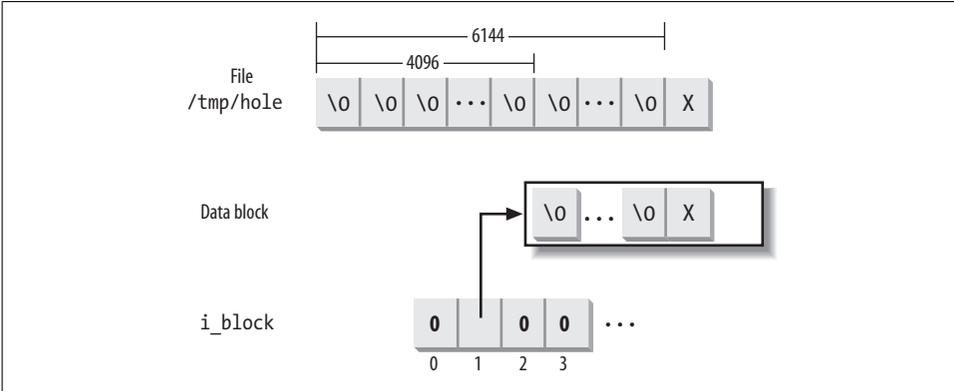
Figure 18-6. A file with an initial hole

Allocating a Data Block

When the kernel has to locate a block holding data for an Ext2 regular file, it invokes the ext2_get_block( ) function. If the block does not exist, the function automatically allocates the block to the file. Remember that this function may be invoked every time the kernel issues a read or write operation on an Ext2 regular file (see the sections "Reading from a File" and "Writing to a File" in Chapter 16); clearly, this function is invoked only if the affected block is not included in the page cache.

The ext2_get_block( ) function handles the data structures already described in the section "Data Blocks Addressing," and when necessary, invokes the ext2_alloc_block( ) function to actually search for a free block in the Ext2 partition. If necessary, the function also allocates the blocks used for indirect addressing (see Figure 18-5).

To reduce file fragmentation, the Ext2 filesystem tries to get a new block for a file near the last block already allocated for the file. Failing that, the filesystem searches for a new block in the block group that includes the file's inode. As a last resort, the free block is taken from one of the other block groups.

The Ext2 filesystem uses preallocation of data blocks. The file does not get only the requested block, but rather a group of up to eight adjacent blocks. The i_prealloc_count field in the ext2_inode_info structure stores the number of data blocks preallocated to a file that are still unused, and the i_prealloc_block field stores the logical

block number of the next preallocated block to be used. All preallocated blocks that remain unused are freed when the file is closed, when it is truncated, or when a write operation is not sequential with respect to the write operation that triggered the block preallocation.

The `ext2_alloc_block( )` function receives as its parameters a pointer to an inode object, a *goal*, and the address of a variable that will store an error code. The goal is a logical block number that represents the preferred position of the new block. The `ext2_get_block( )` function sets the goal parameter according to the following heuristic:

1. If the block that is being allocated and the previously allocated block have consecutive file block numbers, the goal is the logical block number of the previous block plus 1; it makes sense that consecutive blocks as seen by a program should be adjacent on disk.

2. If the first rule does not apply and at least one block has been previously allocated to the file, the goal is one of these blocks' logical block numbers. More precisely, it is the logical block number of the already allocated block that precedes the block to be allocated in the file.

3. If the preceding rules do not apply, the goal is the logical block number of the first block (not necessarily free) in the block group that contains the file's inode.

The `ext2_alloc_block( )` function checks whether the goal refers to one of the preallocated blocks of the file. If so, it allocates the corresponding block and returns its logical block number; otherwise, the function discards all remaining preallocated blocks and invokes `ext2_new_block( )`.

This latter function searches for a free block inside the Ext2 partition with the following strategy:

1. If the preferred block passed to `ext2_alloc_block( )`—the block that is the goal—is free, the function allocates the block.

2. If the goal is busy, the function checks whether one of the next blocks after the preferred block is free.

3. If no free block is found in the near vicinity of the preferred block, the function considers all block groups, starting from the one including the goal. For each block group, the function does the following:

   a. Looks for a group of at least eight adjacent free blocks.

   b. If no such group is found, looks for a single free block.

The search ends as soon as a free block is found. Before terminating, the `ext2_new_block( )` function also tries to preallocate up to eight free blocks adjacent to the free block found and sets the `i_prealloc_block` and `i_prealloc_count` fields of the disk inode to the proper block location and number of blocks.

Releasing a Data Block

When a process deletes a file or truncates it to 0 length, all its data blocks must be reclaimed. This is done by ext2_truncate( ), which receives the address of the file's inode object as its parameter. The function essentially scans the disk inode's i_block array to locate all data blocks and all blocks used for the indirect addressing. These blocks are then released by repeatedly invoking ext2_free_blocks( ).

The ext2_free_blocks( ) function releases a group of one or more adjacent data blocks. Besides its use by ext2_truncate( ), the function is invoked mainly when discarding the preallocated blocks of a file (see the earlier section "Allocating a Data Block"). Its parameters are:

inode
    The address of the inode object that describes the file

block
    The logical block number of the first block to be released

count
    The number of adjacent blocks to be released

The function performs the following actions for each block to be released:

1. Gets the block bitmap of the block group that includes the block to be released

2. Clears the bit in the block bitmap that corresponds to the block to be released and marks the buffer that contains the bitmap as dirty.

3. Increases the bg_free_blocks_count field in the block group descriptor and marks the corresponding buffer as dirty.

4. Increases the s_free_blocks_count field of the disk superblock, marks the corresponding buffer as dirty, and sets the s_dirt flag of the superblock object.

5. If the filesystem has been mounted with the MS_SYNCHRONOUS flag set, it invokes sync_dirty_buffer( ) and waits until the write operation on the bitmap's buffer terminates.

The Ext3 Filesystem

In this section we'll briefly describe the enhanced filesystem that has evolved from Ext2, named *Ext3*. The new filesystem has been designed with two simple concepts in mind:

- To be a journaling filesystem (see the next section)
- To be, as much as possible, compatible with the old Ext2 filesystem

Ext3 achieves both the goals very well. In particular, it is largely based on Ext2, so its data structures on disk are essentially identical to those of an Ext2 filesystem. As a

---

matter of fact, if an Ext3 filesystem has been cleanly unmounted, it can be remounted as an Ext2 filesystem; conversely, creating a journal of an Ext2 filesystem and remounting it as an Ext3 filesystem is a simple, fast operation.

Thanks to the compatibility between Ext3 and Ext2, most descriptions in the previous sections of this chapter apply to Ext3 as well. Therefore, in this section, we focus on the new feature offered by Ext3 — "the journal."

Journaling Filesystems

As disks became larger, one design choice of traditional Unix filesystems (such as Ext2) turns out to be inappropriate. As we know from Chapter 14, updates to filesystem blocks might be kept in dynamic memory for long period of time before being flushed to disk. A dramatic event such as a power-down failure or a system crash might thus leave the filesystem in an inconsistent state. To overcome this problem, each traditional Unix filesystem is checked before being mounted; if it has not been properly unmounted, then a specific program executes an exhaustive, time-consuming check and fixes all the filesystem's data structures on disk.

For instance, the Ext2 filesystem status is stored in the `s_mount_state` field of the superblock on disk. The *e2fsck* utility program is invoked by the boot script to check the value stored in this field; if it is not equal to `EXT2_VALID_FS`, the filesystem was not properly unmounted, and therefore *e2fsck* starts checking all disk data structures of the filesystem.

Clearly, the time spent checking the consistency of a filesystem depends mainly on the number of files and directories to be examined; therefore, it also depends on the disk size. Nowadays, with filesystems reaching hundreds of gigabytes, a single consistency check may take hours. The involved downtime is unacceptable for every production environment or high-availability server.

The goal of a *journaling filesystem* is to avoid running time-consuming consistency checks on the whole filesystem by looking instead in a special disk area that contains the most recent disk write operations named *journal*. Remounting a journaling filesystem after a system failure is a matter of a few seconds.

The Ext3 Journaling Filesystem

The idea behind Ext3 journaling is to perform each high-level change to the filesystem in two steps. First, a copy of the blocks to be written is stored in the journal; then, when the I/O data transfer to the journal is completed (in short, data is *committed to the journal*), the blocks are written in the filesystem. When the I/O data transfer to the filesystem terminates (data is *committed to the filesystem*), the copies of the blocks in the journal are discarded.

While recovering after a system failure, the *e2fsck* program distinguishes the following two cases:

The system failure occurred before a commit to the journal. Either the copies of the blocks relative to the high-level change are missing from the journal or they are incomplete; in both cases, *e2fsck* ignores them.

The system failure occurred after a commit to the journal. The copies of the blocks are valid, and *e2fsck* writes them into the filesystem.

In the first case, the high-level change to the filesystem is lost, but the filesystem state is still consistent. In the second case, *e2fsck* applies the whole high-level change, thus fixing every inconsistency due to unfinished I/O data transfers into the filesystem.

Don't expect too much from a journaling filesystem; it ensures consistency only at the system call level. For instance, a system failure that occurs while you are copying a large file by issuing several `write()` system calls will interrupt the copy operation, thus the duplicated file will be shorter than the original one.

Furthermore, journaling filesystems do not usually copy all blocks into the journal. In fact, each filesystem consists of two kinds of blocks: those containing the so-called *metadata* and those containing regular data. In the case of Ext2 and Ext3, there are six kinds of metadata: superblocks, group block descriptors, inodes, blocks used for indirect addressing (indirection blocks), data bitmap blocks, and inode bitmap blocks. Other filesystems may use different metadata.

Several journaling filesystems, such as SGI's XFS and IBM's JFS, limit themselves to logging the operations affecting metadata. In fact, metadata's log records are sufficient to restore the consistency of the on-disk filesystem data structures. However, since operations on blocks of file data are not logged, nothing prevents a system failure from corrupting the contents of the files.

The Ext3 filesystem, however, can be configured to log the operations affecting both the filesystem metadata and the data blocks of the files. Because logging every kind of write operation leads to a significant performance penalty, Ext3 lets the system administrator decide what has to be logged; in particular, it offers three different journaling modes:

Journal
> All filesystem data and metadata changes are logged into the journal. This mode minimizes the chance of losing the updates made to each file, but it requires many additional disk accesses. For example, when a new file is created, all its data blocks must be duplicated as log records. This is the safest and slowest Ext3 journaling mode.

Ordered
> Only changes to filesystem metadata are logged into the journal. However, the Ext3 filesystem groups metadata and relative data blocks so that data blocks are written to disk *before* the metadata. This way, the chance to have data corruption inside the files is reduced; for instance, each write access that enlarges a file

is guaranteed to be fully protected by the journal. This is the default Ext3 journaling mode.

Writeback

Only changes to filesystem metadata are logged; this is the method found on the other journaling filesystems and is the fastest mode.

The journaling mode of the Ext3 filesystem is specified by an option of the *mount* system command. For instance, to mount an Ext3 filesystem stored in the */dev/sda2* partition on the */jdisk* mount point with the "writeback" mode, the system administrator can type the command:

```
# mount -t ext3 -o data=writeback /dev/sda2 /jdisk
```

The Journaling Block Device Layer

The Ext3 journal is usually stored in a hidden file named *.journal* located in the root directory of the filesystem.

The Ext3 filesystem does not handle the journal on its own; rather, it uses a general kernel layer named *Journaling Block Device*, or *JBD*. Right now, only Ext3 uses the JBD layer, but other filesystems might use it in the future.

The JBD layer is a rather complex piece of software. The Ext3 filesystem invokes the JBD routines to ensure that its subsequent operations don't corrupt the disk data structures in case of system failure. However, JBD typically uses the same disk to log the changes performed by the Ext3 filesystem, and it is therefore vulnerable to system failures as much as Ext3. In other words, JBD must also protect itself from system failures that could corrupt the journal.

Therefore, the interaction between Ext3 and JBD is essentially based on three fundamental units:

Log record

Describes a single update of a disk block of the journaling filesystem.

Atomic operation handle

Includes log records relative to a single high-level change of the filesystem; typically, each system call modifying the filesystem gives rise to a single atomic operation handle.

Transaction

Includes several atomic operation handles whose log records are marked valid for *e2fsck* at the same time.

Log records

A *log record* is essentially the description of a low-level operation that is going to be issued by the filesystem. In some journaling filesystems, the log record consists of exactly the span of bytes modified by the operation, together with the starting position

of the bytes inside the filesystem. The JBD layer, however, uses log records consisting of the whole buffer modified by the low-level operation. This approach may waste a lot of journal space (for instance, when the low-level operation just changes the value of a bit in a bitmap), but it is also much faster because the JBD layer can work directly with buffers and their buffer heads.

Log records are thus represented inside the journal as normal blocks of data (or metadata). Each such block, however, is associated with a small tag of type `journal_block_tag_t`, which stores the logical block number of the block inside the filesystem and a few status flags.

Later, whenever a buffer is being considered by the JBD, either because it belongs to a log record or because it is a data block that should be flushed to disk before the corresponding metadata block (in the "ordered" journaling mode), the kernel attaches a `journal_head` data structure to the buffer head. In this case, the `b_private` field of the buffer head stores the address of the `journal_head` data structure and the `BH_JBD` flag is set (see the section "Block Buffers and Buffer Heads" in Chapter 15).

Atomic operation handles

Every system call modifying the filesystem is usually split into a series of low-level operations that manipulate disk data structures.

For instance, suppose that Ext3 must satisfy a user request to append a block of data to a regular file. The filesystem layer must determine the last block of the file, locate a free block in the filesystem, update the data block bitmap inside the proper block group, store the logical number of the new block either in the file's inode or in an indirect addressing block, write the contents of the new block, and finally, update several fields of the inode. As you see, the append operation translates into many lower-level operations on the data and metadata blocks of the filesystem.

Now, just imagine what could happen if a system failure occurred in the middle of an append operation, when some of the lower-level manipulations have already been executed while others have not. Of course, the scenario could be even worse, with high-level operations affecting two or more files (for example, moving a file from one directory to another).

To prevent data corruption, the Ext3 filesystem must ensure that each system call is handled in an atomic way. An *atomic operation handle* is a set of low-level operations on the disk data structures that correspond to a single high-level operation. When recovering from a system failure, the filesystem ensures that either the whole high-level operation is applied or none of its low-level operations is.

Each atomic operation handle is represented by a descriptor of type `handle_t`. To start an atomic operation, the Ext3 filesystem invokes the `journal_start()` JBD function, which allocates, if necessary, a new atomic operation handle and inserts it into the current transactions (see the next section). Because every low-level operation on the disk might suspend the process, the address of the active handle is stored in the

`journal_info` field of the process descriptor. To notify that an atomic operation is completed, the Ext3 filesystem invokes the `journal_stop()` function.

Transactions

For reasons of efficiency, the JBD layer manages the journal by grouping the log records that belong to several atomic operation handles into a single *transaction*. Furthermore, all log records relative to a handle must be included in the same transaction.

All log records of a transaction are stored in consecutive blocks of the journal. The JBD layer handles each transaction as a whole. For instance, it reclaims the blocks used by a transaction only after all data included in its log records is committed to the filesystem.

As soon as it is created, a transaction may accept log records of new handles. The transaction stops accepting new handles when either of the following occurs:

- A fixed amount of time has elapsed, typically 5 seconds.
- There are no free blocks in the journal left for a new handle.

A transaction is represented by a descriptor of type `transaction_t`. The most important field is `t_state`, which describes the current status of the transaction.

Essentially, a transaction can be:

Complete
All log records included in the transaction have been physically written onto the journal. When recovering from a system failure, *e2fsck* considers every complete transaction of the journal and writes the corresponding blocks into the filesystem. In this case, the `t_state` field stores the value `T_FINISHED`.

Incomplete
At least one log record included in the transaction has not yet been physically written to the journal, or new log records are still being added to the transaction. In case of system failure, the image of the transaction stored in the journal is likely not up-to-date. Therefore, when recovering from a system failure, *e2fsck* does not trust the incomplete transactions in the journal and skips them. In this case, the `t_state` field stores one of the following values:

`T_RUNNING`
Still accepting new atomic operation handles.

`T_LOCKED`
Not accepting new atomic operation handles, but some of them are still unfinished.

`T_FLUSH`
All atomic operation handles have finished, but some log records are still being written to the journal.

`T_COMMIT`
> All log records of the atomic operation handles have been written to disk, but the transaction has yet to be marked as completed on the journal.

At any time the journal may include several transactions, but only one of them is in the `T_RUNNING` state—it is the *active transaction* that is accepting the new atomic operation handle requests issued by the Ext3 filesystem.

Several transactions in the journal might be incomplete, because the buffers containing the relative log records have not yet been written to the journal.

If a transaction is complete, all its log records have been written to the journal but some of the corresponding buffers have yet to be written onto the filesystem. A complete transaction is deleted from the journal when the JBD layer verifies that all buffers described by the log records have been successfully written onto the Ext3 filesystem.

How Journaling Works

Let's try to explain how journaling works with an example: the Ext3 filesystem layer receives a request to write some data blocks of a regular file.

As you might easily guess, we are not going to describe in detail every single operation of the Ext3 filesystem layer and of the JBD layer. There would be far too many issues to be covered! However, we describe the essential actions:

1. The service routine of the `write()` system call triggers the `write` method of the file object associated with the Ext3 regular file. For Ext3, this method is implemented by the `generic_file_write()` function, already described in the section "Writing to a File" in Chapter 16.

2. The `generic_file_write()` function invokes the `prepare_write` method of the `address_space` object several times, once for every page of data involved by the write operation. For Ext3, this method is implemented by the `ext3_prepare_write()` function.

3. The `ext3_prepare_write()` function starts a new atomic operation by invoking the `journal_start()` JBD function. The handle is added to the active transaction. Actually, the atomic operation handle is created only when executing the first invocation of the `journal_start()` function. Following invocations verify that the `journal_info` field of the process descriptor is already set and use the referenced handle.

4. The `ext3_prepare_write()` function invokes the `block_prepare_write()` function already described in Chapter 16, passing to it the address of the `ext3_get_block()` function. Remember that `block_prepare_write()` takes care of preparing the buffers and the buffer heads of the file's page.

5. When the kernel must determine the logical number of a block of the Ext3 file-system, it executes the `ext3_get_block()` function. This function is actually simi-lar to `ext2_get_block()`, which is described in the earlier section "Allocating a Data Block." A crucial difference, however, is that the Ext3 filesystem invokes functions of the JBD layer to ensure that the low-level operations are logged:

   • *Before* issuing a low-level write operation on a metadata block of the filesys-tem, the function invokes `journal_get_write_access()`. Basically, this latter function adds the metadata buffer to a list of the active transaction. How-ever, it must also check whether the metadata is included in an older incom-plete transaction of the journal; in this case, it duplicates the buffer to make sure that the older transactions are committed with the old content.

   • *After* updating the buffer containing the metadata block, the Ext3 filesystem invokes `journal_dirty_metadata()` to move the metadata buffer to the proper dirty list of the active transaction and to log the operation in the journal.

   Notice that metadata buffers handled by the JBD layer are not usually included in the dirty lists of buffers of the inode, so they are not written to disk by the normal disk cache flushing mechanisms described in Chapter 15.

6. If the Ext3 filesystem has been mounted in "journal" mode, the `ext3_prepare_write()` function also invokes `journal_get_write_access()` on every buffer touched by the write operation.

7. Control returns to the `generic_file_write()` function, which updates the page with the data stored in the User Mode address space and then invokes the `commit_write` method of the `address_space` object. For Ext3, the function that implements this method depends on how the Ext3 filesystem has been mounted:

   • If the Ext3 filesystem has been mounted in "journal" mode, the `commit_write` method is implemented by the `ext3_journalled_commit_write()` func-tion, which invokes `journal_dirty_metadata()` on every buffer of data (not metadata) in the page. This way, the buffer is included in the proper dirty list of the active transaction and not in the dirty list of the owner inode; moreover, the corresponding log records are written to the journal. Finally, `ext3_journalled_commit_write()` invokes `journal_stop()` to notify the JBD layer that the atomic operation handle is closed.

   • If the Ext3 filesystem has been mounted in "ordered" mode, the `commit_write` method is implemented by the `ext3_ordered_commit_write()` func-tion, which invokes the `journal_dirty_data()` function on every buffer of data in the page to insert the buffer in a proper list of the active transac-tions. The JBD layer ensures that all buffers in this list are written to disk before the metadata buffers of the transaction. No log record is written onto the journal. Next, `ext3_ordered_commit_write()` executes the normal `generic_commit_write()` function described in Chapter 15, which inserts the

data buffers in the list of the dirty buffers of the owner inode. Finally, `ext3_ordered_commit_write()` invokes `journal_stop()` to notify the JBD layer that the atomic operation handle is closed.

- If the Ext3 filesystem has been mounted in "writeback" mode, the `commit_write` method is implemented by the `ext3_writeback_commit_write()` function, which executes the normal `generic_commit_write()` function described in Chapter 15, which inserts the data buffers in the list of the dirty buffers of the owner inode. Then, `ext3_writeback_commit_write()` invokes `journal_stop()` to notify the JBD layer that the atomic operation handle is closed.

8. The service routine of the `write()` system call terminates here. However, the JBD layer has not finished its work. Eventually, our transaction becomes complete when all its log records have been physically written to the journal. Then `journal_commit_transaction()` is executed.

9. If the Ext3 filesystem has been mounted in "ordered" mode, the `journal_commit_transaction()` function activates the I/O data transfers for all data buffers included in the list of the transaction and waits until all data transfers terminate.

10. The `journal_commit_transaction()` function activates the I/O data transfers for all metadata buffers included in the transaction (and also for all data buffers, if Ext3 was mounted in "journal" mode).

11. Periodically, the kernel activates a checkpoint activity for every complete transaction in the journal. The checkpoint basically involves verifying whether the I/O data transfers triggered by `journal_commit_transaction()` have successfully terminated. If so, the transaction can be deleted from the journal.

Of course, the log records in the journal never play an active role until a system failure occurs. Only during system reboot does the *e2fsck* utility program scan the journal stored in the filesystem and reschedule all write operations described by the log records of the complete transactions.

Process Communication

This chapter explains how User Mode processes can synchronize their actions and exchange data. We already covered several synchronization topics in Chapter 5, but the actors there were kernel control paths, not User Mode programs. We are now ready, after having discussed I/O management and filesystems at length, to extend the discussion to User Mode processes. These processes must rely on the kernel to facilitate interprocess synchronization and communication.

As we saw in the section "Linux File Locking" in Chapter 12, a form of synchronization among User Mode processes can be achieved by creating a (possibly empty) file and using suitable VFS system calls to lock and unlock it. While processes can similarly share data via temporary files protected by locks, this approach is costly because it requires accesses to the filesystem on disk. For this reason, all Unix kernels include a set of system calls that supports process communication without interacting with the filesystem; furthermore, several wrapper functions were developed and inserted in suitable libraries to expedite how processes issue their synchronization requests to the kernel.

As usual, application programmers have a variety of needs that call for different communication mechanisms. Here are the basic mechanisms that Unix systems offer to allow interprocess communication:

Pipes and FIFOs (named pipes)
> Best suited to implement producer/consumer interactions among processes. Some processes fill the pipe with data, while others extract data from the pipe. They are covered in the sections "Pipes" and "FIFOs."

Semaphores
> Represent, as the name implies, the User Mode version of the kernel semaphores discussed in the section "Semaphores" in Chapter 5. They are described in the section "System V IPC."

Messages

Allow processes to exchange messages (short blocks of data) by reading and writing them in predefined message queues. The Linux kernel offers two different versions of messages: System V IPC messages (covered in the section "System V IPC") and POSIX messages (described in the section "POSIX Message Queues").

Shared memory regions

Allow processes to exchange information via a shared block of memory. In applications that must share large amounts of data, this can be the most efficient form of process communication. They are described in the section "System V IPC."

Sockets

Allow processes on different computers to exchange data through a network. Sockets can also be used as a communication tool for processes located on the same host computer; the X Window System graphic interface, for instance, uses a socket to allow client programs to exchange data with the X server.

Pipes

Pipes are an interprocess communication mechanism that is provided in all flavors of Unix. A *pipe* is a one-way flow of data between processes: all data written by a process to the pipe is routed by the kernel to another process, which can thus read it.

In Unix command shells, pipes can be created by means of the | operator. For instance, the following statement instructs the shell to create two processes connected by a pipe:

    $ ls | more

The standard output of the first process, which executes the *ls* program, is redirected to the pipe; the second process, which executes the *more* program, reads its input from the pipe.

Note that the same results can also be obtained by issuing two commands such as the following:

    $ ls > temp
    $ more < temp

The first command redirects the output of *ls* into a regular file; then the second command forces *more* to read its input from the same file. Of course, using pipes instead of temporary files is usually more convenient due to the following reasons:

• The shell statement is much shorter and simpler.
• There is no need to create temporary regular files, which must be deleted later.

Using a Pipe

Pipes may be considered open files that have no corresponding image in the mounted filesystems. A process creates a new pipe by means of the `pipe( )` system call, which returns a pair of file descriptors; the process may then pass these descriptors to its descendants through `fork( )`, thus sharing the pipe with them. The processes can read from the pipe by using the `read( )` system call with the first file descriptor; likewise, they can write into the pipe by using the `write( )` system call with the second file descriptor.

POSIX defines only *half-duplex pipes*, so even though the `pipe( )` system call returns two file descriptors, each process must close one before using the other. If a two-way flow of data is required, the processes must use two different pipes by invoking `pipe( )` twice.

Several Unix systems, such as System V Release 4, implement full-duplex pipes. In a *full-duplex pipe*, both descriptors can be written into and read from, thus there are two bidirectional channels of information. Linux adopts yet another approach: each pipe's file descriptors are still one-way, but it is not necessary to close one of them before using the other.

Let's resume the previous example. When the command shell interprets the `ls|more` statement, it essentially performs the following actions:

1. Invokes the `pipe( )` system call; let's assume that `pipe( )` returns the file descriptors 3 (the pipe's *read channel*) and 4 (the *write channel*).
2. Invokes the `fork( )` system call twice.
3. Invokes the `close( )` system call twice to release file descriptors 3 and 4.

The first child process, which must execute the *ls* program, performs the following operations:

1. Invokes `dup2(4,1)` to copy file descriptor 4 to file descriptor 1. From now on, file descriptor 1 refers to the pipe's write channel.
2. Invokes the `close( )` system call twice to release file descriptors 3 and 4.
3. Invokes the `execve( )` system call to execute the *ls* program (see the section "The exec Functions" in Chapter 20). The program writes its output to the file that has file descriptor 1 (the standard output); i.e., it writes into the pipe.

The second child process must execute the *more* program; therefore, it performs the following operations:

1. Invokes `dup2(3,0)` to copy file descriptor 3 to file descriptor 0. From now on, file descriptor 0 refers to the pipe's read channel.
2. Invokes the `close( )` system call twice to release file descriptors 3 and 4.

3. Invokes the execve( ) system call to execute *more*. By default, that program reads its input from the file that has file descriptor 0 (the standard input); i.e., it reads from the pipe.

In this simple example, the pipe is used by exactly two processes. Because of its implementation, though, a pipe can be used by an arbitrary number of processes.* Clearly, if two or more processes read or write the same pipe, they must explicitly synchronize their accesses by using file locking (see the section "Linux File Locking" in Chapter 12) or IPC semaphores (see the section "IPC Semaphores" later in this chapter).

Many Unix systems provide, besides the pipe( ) system call, two wrapper functions named popen( ) and pclose( ) that handle all the dirty work usually done when using pipes. Once a pipe has been created by means of the popen( ) function, it can be used with the high-level I/O functions included in the C library (fprintf( ), fscanf( ), and so on.

In Linux, popen( ) and pclose( ) are included in the C library. The popen( ) function receives two parameters: the filename pathname of an executable file and a type string specifying the direction of the data transfer. It returns the pointer to a FILE data structure. The popen( ) function essentially performs the following operations:

1. Creates a new pipe by using the pipe( ) system call.

2. Forks a new process, which in turn executes the following operations:

    a. If type is r, it duplicates the file descriptor associated with the pipe's write channel as file descriptor 1 (standard output); otherwise, if type is w, it duplicates the file descriptor associated with the pipe's read channel as file descriptor 0 (standard input).

    b. Closes the file descriptors returned by pipe( ).

    c. Invokes the execve( ) system call to execute the program specified by filename.

3. If type is r, it closes the file descriptor associated with the pipe's write channel; otherwise, if type is w, it closes the file descriptor associated with the pipe's read channel.

4. Returns the address of the FILE file pointer that refers to whichever file descriptor for the pipe is still open.

After the popen( ) invocation, parent and child can exchange information through the pipe: the parent can read (if type is r) or write (if type is w) data by using the FILE pointer returned by the function. The data is written to the standard output or read from the standard input, respectively, by the program executed by the child process.

---

* Because most shells offer pipes that connect only two processes, applications requiring pipes used by more than two processes must be coded in a programming language such as C.

The `pclose( )` function (which receives the file pointer returned by `popen( )` as its parameter) simply invokes the `wait4( )` system call and waits for the termination of the process created by `popen( )`.

Pipe Data Structures

We now have to start thinking again at the system call level. Once a pipe is created, a process uses the `read( )` and `write( )` VFS system calls to access it. Therefore, for each pipe, the kernel creates an inode object plus two file objects—one for reading and the other for writing. When a process wants to read from or write to the pipe, it must use the proper file descriptor.

When the inode object refers to a pipe, its `i_pipe` field points to a `pipe_inode_info` structure shown in Table 19-1.

Table 19-1. The pipe_inode_info structure

Type	Field	Description
`struct wait_queue *`	`wait`	Pipe/FIFO wait queue
`unsigned int`	`nrbufs`	Number of buffers containing data to be read
`unsigned int`	`curbuf`	Index of first buffer containing data to be read
`struct pipe_buffer [16]`	`bufs`	Array of pipe's buffer descriptors
`struct page *`	`tmp_page`	Pointer to a cached page frame
`unsigned int`	`start`	Read position in current pipe buffer
`unsigned int`	`readers`	Flag for (or number of) reading processes
`unsigned int`	`writers`	Flag for (or number of) writing processes
`unsigned int`	`waiting_writers`	Number of writing processes sleeping in the wait queue
`unsigned int`	`r_counter`	Like `readers`, but used when waiting for a process that reads from the FIFO
`unsigned int`	`w_counter`	Like `writers`, but used when waiting for a process that writes into the FIFO
`struct fasync_struct *`	`fasync_readers`	Used for asynchronous I/O notification via signals
`struct fasync_struct *`	`fasync_writers`	Used for asynchronous I/O notification via signals

Besides one inode and two file objects, each pipe has its own set of *pipe buffers*. Essentially, a pipe buffer is a page frame that contains data written into the pipe and yet to be read. Up to Linux 2.6.10, each pipe had just one pipe buffer. In the 2.6.11 kernel, however, data buffering for pipes (and FIFOs) has been heavily revised, and now each pipe makes use of 16 pipe buffers. This change greatly enhances the performance of User Mode applications that write large chunks of data in a pipe.

The `bufs` field of the `pipe_inode_info` data structure stores an array of 16 `pipe_buffer` objects, each of which describes a pipe buffer. The fields of this object are shown in Table 19-2.

Table 19-2. *The fields of the pipe_buffer object*

Type	Field	Description
struct page *	page	Address of the descriptor of the page frame for the pipe buffer
unsigned int	offset	Current position of the significant data inside the page frame
unsigned int	len	Length of the significant data in the pipe buffer
struct pipe_buf_operations *	ops	Address of a table of methods relative to the pipe buffer (NULL if the pipe buffer is empty)

The `ops` field points to the `anon_pipe_buf_ops` table of the pipe buffer's methods, which is a data structure of type `pipe_buf_operations`. Essentially, the table includes three methods:

map

> Invoked before accessing data in the pipe buffer. It simply invokes `kmap()` on the pipe buffer's page frame, just in case the pipe buffer is stored in high memory (see the section "Kernel Mappings of High-Memory Page Frames" in Chapter 8).

unmap

> Invoked when no longer accessing data in the pipe buffer. It invokes `kunmap()` on the pipe buffer's page frame.

release

> Invoked when a pipe buffer is being released. The method implements a one-page memory cache: the page frame released is not the one storing the buffer, but a cached page frame pointed to by the `tmp_page` field of the `pipe_inode_info` data structure (if not `NULL`). The page frame that stored the buffer becomes the new cached page frame.

The 16 pipe buffers can be seen as a global, circular buffer: writing processes keep adding data to this large buffer, while reading process keep removing them. The number of bytes currently written in all pipe buffers and yet to be read is the so-called *pipe size*. For reasons of efficiency, the data yet to be read can be spread among several partially filled pipe buffers: in fact, each write operation may copy the data in a fresh, empty pipe buffer if the previous pipe buffer has not enough free space to store the new data. Hence, the kernel must keep track of:

- The pipe buffer that includes the next byte to be read, and the corresponding offset inside the page frame. The index of this pipe buffer is stored in the `curbuf` field of the `pipe_inode_info` data structure, while the offset is stored in the `offset` field of the corresponding `pipe_buffer` object.

- The first empty pipe buffer. Its index can be computed by adding (modulo 16) the index of the current pipe buffer, which is stored in the `curbuf` field of the

`pipe_inode_info` data structure, and the number of pipe buffers with significant data, which is stored in the `nrbufs` field.

To avoid race conditions on the pipe's data structures, the kernel makes use of the `i_sem` semaphore included in the inode object.

The pipefs special filesystem

A pipe is implemented as a set of VFS objects, which have no corresponding disk images. In Linux 2.6, these VFS objects are organized into the *pipefs* special filesystem to expedite their handling (see the section "Special Filesystems" in Chapter 12). Because this filesystem has no mount point in the system directory tree, users never see it. However, thanks to *pipefs*, the pipes are fully integrated in the VFS layer, and the kernel can handle them in the same way as named pipes or FIFOs, which truly exist as files recognizable to end users (see the later section "FIFOs").

The `init_pipe_fs()` function, typically executed during kernel initialization, registers the *pipefs* filesystem and mounts it (refer to the discussion in the section "Mounting a Generic Filesystem" in Chapter 12):

```
struct file_system_type pipe_fs_type;
pipe_fs_type.name = "pipefs";
pipe_fs_type.get_sb = pipefs_get_sb;
pipe_fs.kill_sb = kill_anon_super;
register_filesystem(&pipe_fs_type);
pipe_mnt = do_kern_mount("pipefs", 0, "pipefs", NULL);
```

The mounted filesystem object that represents the root directory of *pipefs* is stored in the `pipe_mnt` variable.

Creating and Destroying a Pipe

The `pipe()` system call is serviced by the `sys_pipe()` function, which in turn invokes the `do_pipe()` function. To create a new pipe, `do_pipe()` performs the following operations:

1. Invokes the `get_pipe_inode()` function, which allocates and initializes an inode object for the pipe in the *pipefs* filesystem. In particular, this function executes the following actions:

    a. Allocates a new inode in the *pipefs* filesystem.

    b. Allocates a `pipe_inode_info` data structure and stores its address in the `i_pipe` field of the inode.

    c. Sets the `curbuf` and `nrbufs` fields of the `pipe_inode_info` structure to 0; also, fills with zeros all fields of the pipe buffer objects in the `bufs` array.

    d. Initializes the `r_counter` and `w_counter` fields of the `pipe_inode_info` structure to 1.

    e. Sets the `readers` and `writers` fields of the `pipe_inode_info` structure to 1.

2. Allocates a file object and a file descriptor for the read channel of the pipe, sets the f_flag field of the file object to O_RDONLY, and initializes the f_op field with the address of the read_pipe_fops table.

3. Allocates a file object and a file descriptor for the write channel of the pipe, sets the flag field of the file object to O_WRONLY, and initializes the f_op field with the address of the write_pipe_fops table.

4. Allocates a dentry object and uses it to link the two file objects and the inode object (see the section "The Common File Model" in Chapter 12); then inserts the new inode in the *pipefs* special filesystem.

5. Returns the two file descriptors to the User Mode process.

The process that issues a pipe( ) system call is initially the only process that can access the new pipe, both for reading and writing. To represent that the pipe has both a reader and a writer, the readers and writers fields of the pipe_inode_info data structure are initialized to 1. In general, each of these two fields is set to 1 only if the corresponding pipe's file object is still opened by a process; the field is set to 0 if the corresponding file object has been released, because it is no longer accessed by any process.

Forking a new process does not increase the value of the readers and writers fields, so they never rise above 1;* however, it does increase the value of the usage counters of all file objects still used by the parent process (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3). Thus, the objects are not released even when the parent dies, and the pipe stays open for use by the children.

Whenever a process invokes the close( ) system call on a file descriptor associated with a pipe, the kernel executes the fput( ) function on the corresponding file object, which decreases the usage counter. If the counter becomes 0, the function invokes the release method of the file operations (see the sections "The close( ) System Call" and "Files Associated with a Process" in Chapter 12).

Depending on whether the file is associated with the read or write channel, the release method is implemented by either pipe_read_release( ) or pipe_write_release( ); both functions invoke pipe_release( ), which sets either the readers field or the writers field of the pipe_inode_info structure to 0. The function checks whether both the readers and writers fields are equal to 0; in this case, it invokes the pipe buffer's release method of all pipe buffers, thus releasing to the buddy system all pipe's page frames; moreover, the function releases the cached page frame pointed

---

* As we'll see, the readers and writers fields act as counters instead of flags when associated with FIFOs.

to by the `tmp_page` field. Otherwise, if either the `readers` field or the `writers` field is not zero, the function wakes up the processes sleeping in the pipe's wait queue so they can recognize the change in the pipe state.

Reading from a Pipe

A process wishing to get data from a pipe issues a `read( )` system call, specifying the file descriptor associated with the pipe's reading end. As described in the section "The read( ) and write( ) System Calls" in Chapter 12, the kernel ends up invoking the `read` method found in the file operation table associated with the proper file object. In the case of a pipe, the entry for the read method in the `read_pipe_fops` table points to the `pipe_read( )` function.

The `pipe_read( )` function is quite involved, because the POSIX standard specifies several requirements for the pipe's read operations. Table 19-3 summarizes the expected behavior of a `read( )` system call that requests n bytes from a pipe that has a pipe size (number of bytes in the pipe buffers yet to be read) equal to p.

The system call might block the current process in two cases:

- The pipe buffer is empty when the system call starts.
- The pipe buffer does not include all requested bytes, and a writing process was previously put to sleep while waiting for space in the buffer.

Notice that the read operation can be nonblocking: in this case, it completes as soon as all available bytes (even none) are copied into the user address space.[*]

Notice also that the value 0 is returned by the `read( )` system call only if the pipe is empty and no process is currently using the file object associated with the pipe's write channel.

Table 19-3. Reading n bytes from a pipe

Pipe Size p	At least one writing process			No writing process
	Blocking read		Nonblocking read	
	Sleeping writer	No sleeping writer		
$p = 0$	Copy n bytes and return n, waiting for data when the pipe buffer is empty.	Wait for some data, copy it, and return its size.	Return `-EAGAIN`.	Return 0.
$0 < p < n$		Copy p bytes and return p: 0 bytes are left in the pipe buffer.		
$p \geq n$	Copy n bytes and return n: $p-n$ bytes are left in the pipe buffer.			

---

[*] Nonblocking operations are usually requested by specifying the `O_NONBLOCK` flag in the `open( )` system call. This method does not work for pipes, because they cannot be opened. A process can, however, require a nonblocking operation on a pipe by issuing a `fcntl( )` system call on the corresponding file descriptor.

The function performs the following operations:

1. Acquires the i_sem semaphore of the inode.

2. Determines whether the pipe size is 0 by reading the nrbufs field of the pipe_inode_info structure; if the field is equal to zero, all pipe buffers are empty. In this case, it determines whether the function must return or whether the process must be blocked while waiting until another process writes some data in the pipe (see Table 19-3). The type of I/O operation (blocking or nonblocking) is specified by the O_NONBLOCK flag in the f_flags field of the file object. If the current process must be blocked, the function performs the following actions:

   a. Invokes prepare_to_wait() to add current to the wait queue of the pipe (the wait field of the pipe_inode_info structure).

   b. Releases the inode semaphore.

   c. Invokes schedule().

   d. Once awake, invokes finish_wait() to remove current from the wait queue, acquires again the i_sem inode semaphore, and then jumps back to step 2.

3. Gets the index of the current pipe buffer from the curbuf field of the pipe_inode_info data structure.

4. Executes the map method of the pipe buffer.

5. Copies the requested number of bytes—or the number of available bytes in the pipe buffer, if it is smaller—from the pipe's buffer to the user address space.

6. Executes the unmap method of the pipe buffer.

7. Updates the offset and len fields of the corresponding pipe_buffer object.

8. If the pipe buffer has been emptied (len fields of the pipe_buffer object now equal to zero), it invokes the pipe buffer's release method to free the corresponding page frame, sets the ops field in the pipe_buffer object to NULL, advances the index of the current pipe buffer stored in the curbuf field of the pipe_inode_info data structure, and decreases the counter of nonempty pipe buffers in the nrbufs field.

9. If all requested bytes have been copied, it jumps to step 12.

10. Here not all requested bytes have been copied to the User Mode address space. If the pipe size is greater than zero (nrbufs field of the pipe_inode_info data structure not null), it goes back to step 3.

11. There are no more bytes left in the pipe buffers. If there is at least one writing process currently sleeping (that is, the waiting_writers field of the pipe_inode_info data structure is greater than 0), and the read operation is blocking, it invokes wake_up_interruptible_sync() to wake up all processes sleeping on the pipe's wait queue, and jumps back to step 2.

12. Releases the i_sem semaphore of the inode.

13. Invokes `wake_up_interruptible_sync()` to wake up all writer processes sleeping on the pipe's wait queue.

14. Returns the number of bytes copied into the user address space.

Writing into a Pipe

A process wishing to put data into a pipe issues a `write()` system call, specifying the file descriptor for the writing end of the pipe. The kernel satisfies this request by invoking the `write` method of the proper file object; the corresponding entry in the `write_pipe_fops` table points to the `pipe_write()` function.

Table 19-4 summarizes the behavior, specified by the POSIX standard, of a `write()` system call that requested to write n bytes into a pipe having u unused bytes in its buffer. In particular, the standard requires that write operations involving a small number of bytes must be atomically executed. More precisely, if two or more processes are concurrently writing into a pipe, each write operation involving fewer than 4,096 bytes (the pipe buffer size) must finish without being interleaved with write operations of other processes to the same pipe. However, write operations involving more than 4,096 bytes may be nonatomic and may also force the calling process to sleep.

Table 19-4. Writing n bytes to a pipe

| Available buffer space u | At least one reading process | | No reading process |
	Blocking write	Nonblocking write	
$u<n\leq 4,096$	Wait until $n-u$ bytes are freed, copy n bytes, and return n.	Return `-EAGAIN`.	Send `SIGPIPE` signal and return `-EPIPE`.
$n>4,096$	Copy n bytes (waiting when necessary) and return n.	If $u>0$, copy u bytes and return u; return `-EAGAIN`.	
$u\geq n$	Copy n bytes and return n.		

Moreover, each write operation to a pipe must fail if the pipe does not have a reading process (that is, if the `readers` field of the pipe's inode object has the value 0). In this case, the kernel sends a `SIGPIPE` signal to the writing process and terminates the `write()` system call with the `-EPIPE` error code, which usually leads to the familiar "Broken pipe" message.

The `pipe_write()` function performs the following operations:

1. Acquires the `i_sem` semaphore of the inode.

2. Checks whether the pipe has at least one reading process. If not, it sends a `SIGPIPE` signal to the `current` process, releases the inode semaphore, and returns an `-EPIPE` value.

3. Determines the index of the last written pipe buffers by adding the `curbuf` and `nrbufs` fields of the `pipe_inode_info` data structure and subtracting 1. If this pipe buffer has enough free space to store all the bytes to be written, then it copies the data into it:

    a. Executes the `map` method of the pipe buffer.

    b. Copies all the bytes in the pipe buffer.

    c. Executes the `unmap` method of the pipe buffer.

    d. Updates the `len` field of the corresponding `pipe_buffer` object.

    e. Jumps to step 11.

4. If the `nrbufs` field of the `pipe_inode_info` data structure is equal to 16, there is no empty pipe buffer to store the bytes (yet) to be written. In this case:

    a. If the write operation is nonblocking, it jumps to step 11 to terminate by returning the `-EAGAIN` error code.

    b. If the write operation is blocking, it adds 1 to the `waiting_writers` field of the `pipe_inode_info` structure, invokes `prepare_to_wait()` to add `current` to the wait queue of the pipe (the `wait` field of the `pipe_inode_info` structure), releases the inode semaphore, and invokes `schedule()`. Once awake, it invokes `finish_wait()` to remove `current` from the wait queue, again acquires the inode semaphore, decreases the `waiting_writers` field, and then jumps back to step 4.

5. Now there is at least one empty pipe buffer. Determines the index of the first empty pipe buffer by adding the `curbuf` and `nrbufs` fields of the `pipe_inode_info` data structure.

6. Allocates a new page frame from the buddy system, unless the `tmp_page` field of the `pipe_inode_info` data structure is not `NULL`.

7. Copies up to 4,096 bytes from the User Mode address space into the page frame (temporarily mapping it in the Kernel Mode linear address space, if necessary).

8. Updates the fields of the `pipe_buffer` object associated with the pipe buffer by setting the `page` field to the address of the page frame descriptor, the `ops` field to the address of the `anon_pipe_buf_ops` table, the `offset` field to 0, and the `len` field to the number of written bytes.

9. Increases the counter of nonempty pipe buffers stored in the `nrbufs` field of the `pipe_inode_info` data structure.

10. If not all requested bytes were written, it jumps back to step 4.

11. Releases the inode semaphore.

12. Wakes up all reader processes sleeping on the pipe's wait queue.

13. Returns the number of bytes written into the pipe's buffer (or an error code if writing was not possible).

FIFOs

Although pipes are a simple, flexible, and efficient communication mechanism, they have one main drawback—namely, that there is no way to open an already existing pipe. This makes it impossible for two arbitrary processes to share the same pipe, unless the pipe was created by a common ancestor process.

This drawback is substantial for many application programs. Consider, for instance, a database engine server, which continuously polls client processes wishing to issue some queries and which sends the results of the database lookups back to them. Each interaction between the server and a given client might be handled by a pipe. However, client processes are usually created on demand by a command shell when a user explicitly queries the database; server and client processes thus cannot easily share a pipe.

To address such limitations, Unix systems introduce a special file type called a *named pipe* or *FIFO* (which stands for "first in, first out;" the first byte written into the special file is also the first byte that is read). Each FIFO is much like a pipe: rather than owning disk blocks in the filesystems, an opened FIFO is associated with a kernel buffer that temporarily stores the data exchanged by two or more processes.

Thanks to the disk inode, however, a FIFO can be accessed by every process, because the FIFO filename is included in the system's directory tree. Thus, in our example, the communication between server and clients may be easily established by using FIFOs instead of pipes. The server creates, at startup, a FIFO used by client programs to make their requests. Each client program creates, before establishing the connection, another FIFO to which the server program can write the answer to the query and includes the FIFO's name in the initial request to the server.

In Linux 2.6, FIFOs and pipes are almost identical and use the same `pipe_inode_info` structures. As a matter of fact, the `read` and `write` file operation methods of a FIFO are implemented by the same `pipe_read()` and `pipe_write()` functions described in the earlier sections "Reading from a Pipe" and "Writing into a Pipe." Actually, there are only two significant differences:

- FIFO inodes appear on the system directory tree rather than on the *pipefs* special filesystem.
- FIFOs are a bidirectional communication channel; that is, it is possible to open a FIFO in read/write mode.

To complete our description, therefore, we just have to explain how FIFOs are created and opened.

Creating and Opening a FIFO

A process creates a FIFO by issuing a `mknod()`* system call (see the section "Device Files" in Chapter 13), passing to it as parameters the pathname of the new FIFO and the value `S_IFIFO` (`0x10000`) logically ORed with the permission bit mask of the new file. POSIX introduces a function named `mkfifo()` specifically to create a FIFO. This call is implemented in Linux, as in System V Release 4, as a C library function that invokes `mknod()`.

Once created, a FIFO can be accessed through the usual `open()`, `read()`, `write()`, and `close()` system calls, but the VFS handles it in a special way, because the FIFO inode and file operations are customized and do not depend on the filesystems in which the FIFO is stored.

The POSIX standard specifies the behavior of the `open()` system call on FIFOs; the behavior depends essentially on the requested access type, the kind of I/O operation (blocking or nonblocking), and the presence of other processes accessing the FIFO.

A process may open a FIFO for reading, for writing, or for reading and writing. The file operations associated with the corresponding file object are set to special methods for these three cases.

When a process opens a FIFO, the VFS performs the same operations as it does for device files (see the section "VFS Handling of Device Files" in Chapter 13). The inode object associated with the opened FIFO is initialized by a filesystem-dependent `read_inode` superblock method; this method always checks whether the inode on disk represents a special file, and invokes, if necessary, the `init_special_inode()` function. In turn, this function sets the `i_fop` field of the inode object to the address of the `def_fifo_fops` table. Later, the kernel sets the file operation table of the file object to `def_fifo_fops`, and executes its `open` method, which is implemented by `fifo_open()`.

The `fifo_open()` function initializes the data structures specific to the FIFO; in particular, it performs the following operations:

1. Acquires the `i_sem` inode semaphore.
2. Checks the `i_pipe` field of the inode object; if it is `NULL`, it allocates and it initializes a new `pipe_inode_info` structure, as in steps 1b–1e in the earlier section "Creating and Destroying a Pipe."
3. Depending on the access mode specified as the parameter of the `open()` system call, it initializes the `f_op` field of the file object with the address of the proper file operation table (see Table 19-5).

---

* In fact, `mknod()` can be used to create nearly every kind of file, such as block and character device files, FIFOs, and even regular files (it cannot create directories or sockets, though).

Table 19-5. FIFO's file operations

Access type	File operations	Read method	Write method
Read-only	`read_fifo_fops`	`pipe_read( )`	`bad_pipe_w( )`
Write-only	`write_fifo_fops`	`bad_pipe_r( )`	`pipe_write( )`
Read/write	`rdwr_fifo_fops`	`pipe_read( )`	`pipe_write( )`

4. If the access mode is either read-only or read/write, it adds one to the `readers` and `r_counter` fields of the `pipe_inode_info` structure. Moreover, if the access mode is read-only and there is no other reading process, it wakes up any writing process sleeping in the wait queue.

5. If the access mode is either write-only or read/write, it adds one to the `writers` and `w_counter` fields of the `pipe_inode_info` structure. Moreover, if the access mode is write-only and there is no other writing process, it wakes up any reading process sleeping in the wait queue.

6. If there are no readers or no writers, it decides whether the function should block or terminate returning an error code (see Table 19-6).

Table 19-6. Behavior of the fifo_open( ) function

Access type	Blocking	Nonblocking
Read-only, with writers	Successfully return	Successfully return
Read-only, no writer	Wait for a writer	Successfully return
Write-only, with readers	Successfully return	Successfully return
Write-only, no reader	Wait for a reader	Return -ENXIO
Read/write	Successfully return	Successfully return

7. Releases the inode semaphore, and terminates, returning 0 (success).

The FIFO's three specialized file operation tables differ mainly in the implementation of the `read` and `write` methods. If the access type allows read operations, the `read` method is implemented by the `pipe_read( )` function. Otherwise, it is implemented by `bad_pipe_r( )`, which only returns an error code. Similarly, if the access type allows write operations, the `write` method is implemented by the `pipe_write( )` function; otherwise, it is implemented by `bad_pipe_w( )`, which also returns an error code.

System V IPC

IPC is an abbreviation for Interprocess Communication and commonly refers to a set of mechanisms that allow a User Mode process to do the following:

- Synchronize itself with other processes by means of semaphores
- Send messages to other processes or receive messages from them
- Share a memory area with other processes

System V IPC first appeared in a development Unix variant called "Columbus Unix" and later was adopted by AT&T's System III. It is now found in most Unix systems, including Linux.

IPC data structures are created dynamically when a process requests an *IPC resource* (a semaphore, a message queue, or a shared memory region). An IPC resource is persistent: unless explicitly removed by a process, it is kept in memory and remains available until the system is shut down. An IPC resource may be used by every process, including those that do not share the ancestor that created the resource.

Because a process may require several IPC resources of the same type, each new resource is identified by a 32-bit *IPC key*, which is similar to the file pathname in the system's directory tree. Each IPC resource also has a 32-bit *IPC identifier*, which is somewhat similar to the file descriptor associated with an open file. IPC identifiers are assigned to IPC resources by the kernel and are unique within the system, while IPC keys can be freely chosen by programmers.

When two or more processes wish to communicate through an IPC resource, they all refer to the IPC identifier of the resource.

Using an IPC Resource

IPC resources are created by invoking the semget( ), msgget( ), or shmget( ) functions, depending on whether the new resource is a semaphore, a message queue, or a shared memory region.

The main objective of each of these three functions is to derive from the IPC key (passed as the first parameter) the corresponding IPC identifier, which is then used by the process for accessing the resource. If there is no IPC resource already associated with the IPC key, a new resource is created. If everything goes right, the function returns a positive IPC identifier; otherwise, it returns one of the error codes listed in Table 19-7.

Table 19-7. Error codes returned while requesting an IPC identifier

Error code	Description
EACCESS	Process does not have proper access rights
EEXIST	Process tried to create an IPC resource with the same key as one that already exists
EINVAL	Invalid argument in a parameter of semget( ), msgget( ), or shmget( )
ENOENT	No IPC resource with the requested key exists and the process did not ask to create it
ENOMEM	No more storage is left for an additional IPC resource
ENOSPC	Maximum limit on the number of IPC resources has been exceeded

Assume that two independent processes want to share a common IPC resource. This can be achieved in two possible ways:

- The processes agree on some fixed, predefined IPC key. This is the simplest case, and it works quite well for every complex application implemented by many processes. However, there's a chance that the same IPC key is chosen by another unrelated program. In this case, the IPC functions might be successfully invoked and still return the IPC identifier of the wrong resource.[*]

- One process issues a `semget()`, `msgget()`, or `shmget()` function by specifying `IPC_PRIVATE` as its IPC key. A new IPC resource is thus allocated, and the process can either communicate its IPC identifier to the other process in the application[†] or fork the other process itself. This method ensures that the IPC resource cannot be used accidentally by other applications.

The last parameter of the `semget()`, `msgget()`, and `shmget()` functions can include three flags. `IPC_CREAT` specifies that the IPC resource must be created, if it does not already exist; `IPC_EXCL` specifies that the function must fail if the resource already exists and the `IPC_CREAT` flag is set; `IPC_NOWAIT` specifies that the process should never block when accessing the IPC resource (typically, when fetching a message or when acquiring a semaphore).

Even if the process uses the `IPC_CREAT` and `IPC_EXCL` flags, there is no way to ensure exclusive access to an IPC resource, because other processes may always refer to the resource by using its IPC identifier.

To minimize the risk of incorrectly referencing the wrong resource, the kernel does not recycle IPC identifiers as soon as they become free. Instead, the IPC identifier assigned to a resource is almost always larger than the identifier assigned to the previously allocated resource of the same type. (The only exception occurs when the 32-bit IPC identifier overflows.) Each IPC identifier is computed by combining a *slot usage sequence number* relative to the resource type, an arbitrary *slot index* for the allocated resource, and an arbitrary value chosen in the kernel that is greater than the maximum number of allocatable resources. If we choose s to represent the slot usage sequence number, M to represent the upper bound on the number of allocatable resources, and i to represent the slot index, where $0 \leq i < M$, each IPC resource's ID is computed as follows:

IPC identifier $= s \times M + i$

In Linux 2.6, the value of M is set to 32,768 (`IPCMNI` macro). The slot usage sequence number s is initialized to 0 and is increased by 1 at every resource allocation. When s reaches a predefined threshold, which depends on the type of IPC resource, it restarts from 0.

---

[*] The `ftok()` function attempts to create a new key from a file pathname and an 8-bit project identifier passed as its parameters. It does not guarantee, however, a unique key number, because there is a small chance that it will return the same IPC key to two different applications using different pathnames and project identifiers.

[†] This implies, of course, the existence of another communication channel between the processes not based on IPC.

Every type of IPC resource (semaphores, message queues, and shared memory areas) owns an `ipc_ids` data structure, which includes the fields shown in Table 19-8.

Table 19-8. The fields of the ipc_ids data structure

Type	Field	Description
int	in_use	Number of allocated IPC resources
int	max_id	Maximum slot index in use
unsigned short	seq	Slot usage sequence number for the next allocation
unsigned short	seq_max	Maximum slot usage sequence number
struct semaphore	sem	Semaphore protecting the `ipc_ids` data structure
struct ipc_id_ary	nullentry	Fake data structure pointed to by the `entries` field if this IPC resource cannot be initialized (normally not used)
struct ipc_id_ary *	entries	Pointer to the `ipc_id_ary` data structure for this resource

The `ipc_id_ary` data structure consists of two fields: `p` and `size`. The `p` field is an array of pointers to `kern_ipc_perm` data structures, one for every allocatable resource. The `size` field is the size of this array. Initially, the array stores 1, 16, or 128 pointers, respectively for shared memory regions, message queues, and semaphores. The kernel dynamically increases the size of the array when it becomes too small. However, there is an upper bound on the number of resources for each given type. The system administrator may change these bounds by writing into the */proc/sys/kernel/sem*, */proc/sys/kernel/msgmni*, and */proc/sys/kernel/shmmni* files, respectively.

Each `kern_ipc_perm` data structure is associated with an IPC resource and contains the fields shown in Table 19-9. The `uid`, `gid`, `cuid`, and `cgid` fields store the user and group identifiers of the resource's creator and the user and group identifiers of the current resource's owner, respectively. The `mode` bit mask includes six flags, which store the read and write access permissions for the resource's owner, the resource's group, and all other users. IPC access permissions are similar to file access permissions described in the section "Access Rights and File Mode" in Chapter 1, except that the Execute permission flag is not used.

Table 19-9. The fields in the kern_ipc_perm structure

Type	Field	Description
spinlock_t	lock	Spin lock protecting the IPC resource descriptor
int	deleted	Flag set if the resource has been released
int	key	IPC key
unsigned int	uid	Owner user ID
unsigned int	gid	Owner group ID
unsigned int	cuid	Creator user ID
unsigned int	cgid	Creator group ID

Table 19-9. The fields in the kern_ipc_perm structure (continued)

Type	Field	Description
`unsigned short`	`mode`	**Permission bit mask**
`unsigned long`	`seq`	**Slot usage sequence number**
`void *`	`security`	Pointer to a security structure (used by SELinux)

The `kern_ipc_perm` data structure also includes a key field (which contains the IPC key of the corresponding resource) and a `seq` field (which stores the slot usage sequence number *s* used to compute the IPC identifier of the resource).

The `semctl()`, `msgctl()`, and `shmctl()` functions may be used to handle IPC resources. The `IPC_SET` subcommand allows a process to change the owner's user and group identifiers and the permission bit mask in the `ipc_perm` data structure. The `IPC_STAT` and `IPC_INFO` subcommands retrieve some information concerning a resource. Finally, the `IPC_RMID` subcommand releases an IPC resource. Depending on the type of IPC resource, other specialized subcommands are also available.[*]

Once an IPC resource is created, a process may act on the resource by means of a few specialized functions. A process may acquire or release an IPC semaphore by issuing the `semop()` function. When a process wants to send or receive an IPC message, it uses the `msgsnd()` and `msgrcv()` functions, respectively. Finally, a process attaches and detaches an IPC shared memory region in its address space by means of the `shmat()` and `shmdt()` functions, respectively.

The ipc( ) System Call

All IPC functions must be implemented through suitable Linux system calls. Actually, in the 80×86 architecture, there is just one IPC system call named `ipc()`. When a process invokes an IPC function, let's say `msgget()`, it really invokes a wrapper function in the C library. This in turn invokes the `ipc()` system call by passing to it all the parameters of `msgget()` plus a proper subcommand code—in this case, `MSGGET`. The `sys_ipc()` service routine examines the subcommand code and invokes the kernel function that implements the requested service.

The `ipc()` "multiplexer" system call is a legacy from older Linux versions, which included the IPC code in a dynamic module (see Appendix B). It did not make much sense to reserve several system call entries in the `system_call` table for a kernel component that could be missing, so the kernel designers adopted the multiplexer approach.

Nowadays, System V IPC can no longer be compiled as a dynamic module, and there is no justification for using a single IPC system call. As a matter of fact, Linux provides

---

[*] An IPC design flaw is that a User Mode process cannot atomically create and initialize an IPC semaphore, because these two operations are performed by two different IPC functions.

one system call for each IPC function on Hewlett-Packard's Alpha architecture and on Intel's IA-64.

IPC Semaphores

IPC semaphores are quite similar to the kernel semaphores introduced in Chapter 5; they are counters used to provide controlled access to shared data structures for multiple processes.

The semaphore value is positive if the protected resource is available, and 0 if the protected resource is currently not available. A process that wants to access the resource tries to decrease the semaphore value; the kernel, however, blocks the process until the operation on the semaphore yields a positive value. When a process relinquishes a protected resource, it increases its semaphore value; in doing so, any other process waiting for the semaphore is woken up.

Actually, IPC semaphores are more complicated to handle than kernel semaphores for two main reasons:

- Each IPC semaphore is a set of one or more semaphore values, not just a single value like a kernel semaphore. This means that the same IPC resource can protect several independent shared data structures. The number of semaphore values in each IPC semaphore must be specified as a parameter of the `semget( )` function when the resource is being allocated. From now on, we'll refer to the counters inside an IPC semaphore as *primitive semaphores*. There are bounds both on the number of IPC semaphore resources (by default, 128) and on the number of primitive semaphores inside a single IPC semaphore resource (by default, 250); however, the system administrator can easily modify these bounds by writing into the */proc/sys/kernel/sem* file.

- System V IPC semaphores provide a fail-safe mechanism for situations in which a process dies without being able to undo the operations that it previously issued on a semaphore. When a process chooses to use this mechanism, the resulting operations are called *undoable* semaphore operations. When the process dies, all of its IPC semaphores can revert to the values they would have had if the process had never started its operations. This can help prevent other processes that use the same semaphores from remaining blocked indefinitely as a consequence of the terminating process failing to manually undo its semaphore operations.

First, we'll briefly sketch the typical steps performed by a process wishing to access one or more resources protected by an IPC semaphore:

1. Invokes the `semget( )` wrapper function to get the IPC semaphore identifier, specifying as the parameter the IPC key of the IPC semaphore that protects the shared resources. If the process wants to create a new IPC semaphore, it also specifies the `IPC_CREATE` or `IPC_PRIVATE` flag and the number of primitive semaphores required (see the section "Using an IPC Resource" earlier in this chapter).

2. Invokes the semop( ) wrapper function to test and decrease all primitive sema-phore values involved. If all the tests succeed, the decrements are performed, the function terminates, and the process is allowed to access the protected resources. If some semaphores are in use, the process is usually suspended until some other process releases the resources. The function receives as its parameters the IPC semaphore identifier, an array of integers specifying the operations to be atomi-cally performed on the primitive semaphores, and the number of such opera-tions. Optionally, the process may specify the SEM_UNDO flag, which instructs the kernel to reverse the operations, should the process exit without releasing the primitive semaphores.

3. When relinquishing the protected resources, it invokes the semop( ) function again to atomically increase all primitive semaphores involved.

4. Optionally, it invokes the semctl( ) wrapper function, specifying the IPC_RMID command to remove the IPC semaphore from the system.

Now we can discuss how the kernel implements IPC semaphores. The data struc-tures involved are shown in Figure 19-1. The sem_ids variable stores the ipc_ids data structure of the IPC semaphore resource type; the corresponding ipc_id_ary data structure contains an array of pointers to sem_array data structures, one item for every IPC semaphore resource.
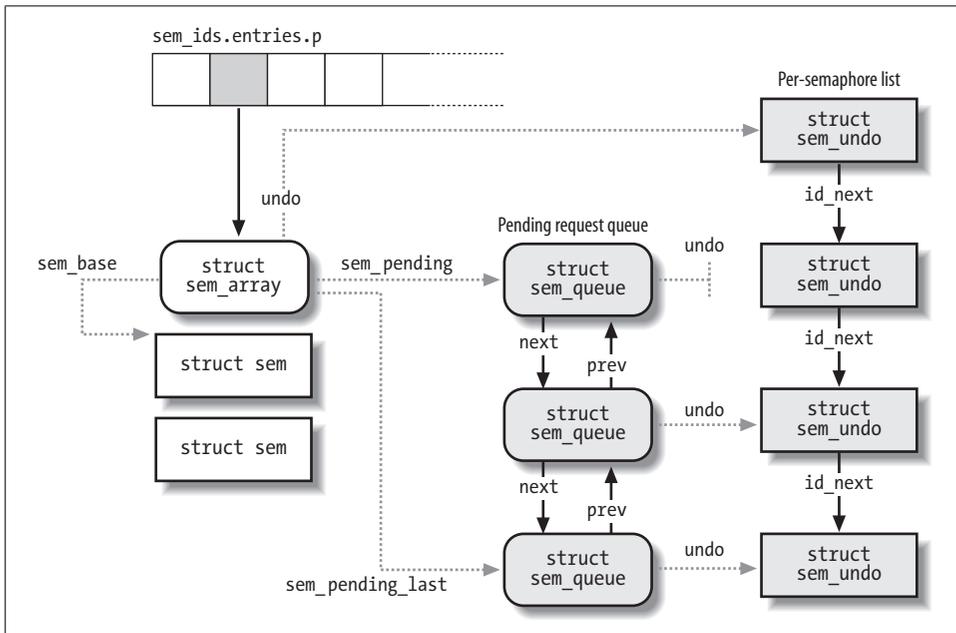
Figure 19-1. IPC semaphore data structures

Formally, the array stores pointers to `kern_ipc_perm` data structures, but each structure is simply the first field of the `sem_array` data structure. All fields of the `sem_array` data structure are shown in Table 19-10.

Table 19-10. The fields in the sem_array data structure

Type	Field	Description
struct kern_ipc_perm	sem_perm	`kern_ipc_perm` data structure
long	sem_otime	Timestamp of last `semop()`
long	sem_ctime	Timestamp of last change
struct sem *	sem_base	Pointer to first `sem` structure
struct sem_queue *	sem_pending	Pending operations
struct sem_queue **	sem_pending_last	Last pending operation
struct sem_undo *	undo	Undo requests
unsigned long	sem_nsems	Number of semaphores in array

The `sem_base` field points to an array of `sem` data structures, one for every IPC primitive semaphore. The latter data structure includes only two fields:

semval
> The value of the semaphore's counter.

sempid
> The PID of the last process that accessed the semaphore. This value can be queried by a process through the `semctl()` wrapper function.

Undoable semaphore operations

If a process aborts suddenly, it cannot undo the operations that it started (for instance, release the semaphores it reserved); so by declaring them undoable, the process lets the kernel return the semaphores to a consistent state and allow other processes to proceed. Processes can request undoable operations by specifying the `SEM_UNDO` flag in the `semop()` function.

Information to help the kernel reverse the undoable operations performed by a given process on a given IPC semaphore resource is stored in a `sem_undo` data structure. It essentially contains the IPC identifier of the semaphore and an array of integers representing the changes to the primitive semaphore's values caused by all undoable operations performed by the process.

A simple example can illustrate how such `sem_undo` elements are used. Consider a process that uses an IPC semaphore resource containing four primitive semaphores. Suppose that it invokes the `semop()` function to increase the first counter by 1 and decrease the second by 2. If it specifies the `SEM_UNDO` flag, the integer in the first array element in the `sem_undo` data structure is decreased by 1, the integer in the second element is increased by 2, and the other two integers are left unchanged. Further

undoable operations on the IPC semaphore performed by the same process change the integers stored in the sem_undo structure accordingly. When the process exits, any nonzero value in that array corresponds to one or more unbalanced operations on the corresponding primitive semaphore; the kernel reverses these operations, simply adding the nonzero value to the corresponding semaphore's counter. In other words, the changes made by the aborted process are backed out while the changes made by other processes are still reflected in the state of the semaphores.

For each process, the kernel keeps track of all semaphore resources handled with undoable operations so that it can roll them back if the process unexpectedly exits. Furthermore, for each semaphore, the kernel has to keep track of all its sem_undo structures so it can quickly access them whenever a process uses semctl( ) to force an explicit value into a primitive semaphore's counter or to destroy an IPC semaphore resource.

The kernel is able to handle these tasks efficiently, thanks to two lists, which we denote as the *per-process* and the *per-semaphore* lists. The first list keeps track of all semaphores operated upon by a given process with undoable operations. The second list keeps track of all processes that are acting on a given semaphore with undoable operations. More precisely:

- The per-process list includes all sem_undo data structures corresponding to IPC semaphores on which the process has performed undoable operations. The sysvsem.undo_list field of the process descriptor points to a data structure, of type sem_undo_list, which in turn contains a pointer to the first element of the list; the proc_next field of each sem_undo data structure points to the next element in the list. (As mentioned in the section "The clone(), fork(), and vfork() System Calls" in Chapter 3, clone processes created by passing the CLONE_SYSVSEM flag to the clone( ) system call share the same list of undoable semaphore operations, because they share the same sem_undo_list descriptor.)

- The per-semaphore list includes all sem_undo data structures corresponding to the processes that performed undoable operations on the semaphore. The undo field of the sem_array data structure points to the first element of the list, while the id_next field of each sem_undo data structure points to the next element in the list.

The per-process list is used when a process terminates. The exit_sem( ) function, which is invoked by do_exit( ), walks through the list and reverses the effect of any unbalanced operation for every IPC semaphore touched by the process. By contrast, the per-semaphore list is mainly used when a process invokes the semctl( ) function to force an explicit value into a primitive semaphore. The kernel sets the corresponding element to 0 in the arrays of all sem_undo data structures referring to that IPC semaphore resource, because it would no longer make any sense to reverse the effect of previous undoable operations performed on that primitive semaphore. Moreover,

the per-semaphore list is also used when an IPC semaphore is destroyed; all related
sem_undo data structures are invalidated by setting the semid field to −1.[*]

The queue of pending requests

The kernel associates a *queue of pending requests* with each IPC semaphore to iden-
tify processes that are waiting on one (or more) of the semaphores in the array. The
queue is a doubly linked list of sem_queue data structures whose fields are shown in
Table 19-11. The first and last pending requests in the queue are referenced, respec-
tively, by the sem_pending and sem_pending_last fields of the sem_array structure.
This last field allows the list to be handled as easily as a FIFO; new pending requests
are added to the end of the list so they will be serviced later. The most important
fields of a pending request are nsops (which stores the number of primitive sema-
phores involved in the pending operation) and sops (which points to an array of inte-
ger values describing each semaphore operation). The sleeper field stores the
descriptor address of the sleeping process that requested the operation.

Table 19-11. The fields in the sem_queue data structure

Type	Field	Description
struct sem_queue *	next	Pointer to next queue element
struct sem_queue **	prev	Pointer to previous queue element
struct task_struct *	sleeper	Pointer to the sleeping process that requested the semaphore opera-tion
struct sem_undo *	undo	Pointer to sem_undo structure
int	pid	Process identifier
int	status	Completion status of operation
struct sem_array *	sma	Pointer to IPC semaphore descriptor
int	id	Slot index of the IPC semaphore resource
struct sembuf *	sops	Pointer to array of pending operations
int	nsops	Number of pending operations
int	alter	Flag denoting whether the operation modifies the semaphore array

Figure 19-1 illustrates an IPC semaphore that has three pending requests. The sec-
ond and third requests refer to undoable operations, so the undo field of the sem_queue
data structure points to the corresponding sem_undo structure; the first pending
request has a NULL undo field because the corresponding operation is not undoable.

---

[*] Notice that they are just invalidated and not freed, because it would be too costly to remove the data struc-
tures from the per-process lists of all processes.

IPC Messages

Processes can communicate with one another by means of IPC messages. Each message generated by a process is sent to an *IPC message queue*, where it stays until another process reads it.

A message is composed of a fixed-size *header* and a variable-length *text*; it can be labeled with an integer value (the *message type*), which allows a process to selectively retrieve messages from its message queue.* Once a process has read a message from an IPC message queue, the kernel destroys the message; therefore, only one process can receive a given message.

To send a message, a process invokes the `msgsnd( )` function, passing the following as parameters:

- The IPC identifier of the destination message queue
- The size of the message text
- The address of a User Mode buffer that contains the message type immediately followed by the message text

To retrieve a message, a process invokes the `msgrcv( )` function, passing to it:

- The IPC identifier of the IPC message queue resource
- The pointer to a User Mode buffer to which the message type and message text should be copied
- The size of this buffer
- A value *t* that specifies what message should be retrieved

If the value *t* is 0, the first message in the queue is returned. If *t* is positive, the first message in the queue with its type equal to *t* is returned. Finally, if *t* is negative, the function returns the first message whose message type is the lowest value less than or equal to the absolute value of *t*.

To avoid resource exhaustion, there are some limits on the number of IPC message queue resources allowed (by default, 16), on the size of each message (by default, 8,192 bytes), and on the maximum total size of the messages in a queue (by default, 16,384 bytes). As usual, however, the system administrator can tune these values by writing into the */proc/sys/kernel/msgmni*, */proc/sys/kernel/msgmnb*, and */proc/sys/kernel/msgmax* files, respectively.

The data structures associated with IPC message queues are shown in Figure 19-2. The `msg_ids` variable stores the `ipc_ids` data structure of the IPC message queue resource type; the corresponding `ipc_id_ary` data structure contains an array of

---

* As we'll see, the message queue is implemented by means of a linked list. Because messages can be retrieved in an order different from "first in, first out," the name "message queue" is not appropriate. However, new messages are always put at the end of the linked list.

pointers to shmid_kernel data structures—one item for every IPC message queue resource. Formally, the array stores pointers to kern_ipc_perm data structures, but each such structure is simply the first field of the msg_queue data structure. All fields of the msg_queue data structure are shown in Table 19-12.
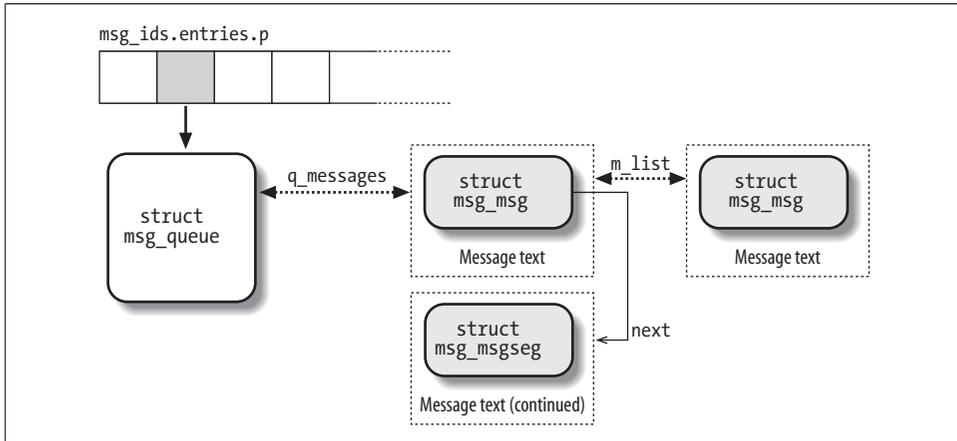
Figure 19-2. IPC message queue data structures

Table 19-12. The msg_queue data structure

Type	Field	Description
struct kern_ipc_perm	q_perm	kern_ipc_perm data structure
long	q_stime	Time of last msgsnd( )
long	q_rtime	Time of last msgrcv( )
long	q_ctime	Last change time
unsigned long	q_qcbytes	Number of bytes in queue
unsigned long	q_qnum	Number of messages in queue
unsigned long	q_qbytes	Maximum number of bytes in queue
int	q_lspid	PID of last msgsnd( )
int	q_lrpid	PID of last msgrcv( )
struct list_head	q_messages	List of messages in queue
struct list_head	q_receivers	List of processes receiving messages
struct list_head	q_senders	List of processes sending messages

The most important field is q_messages, which represents the head (i.e., the first dummy element) of a doubly linked circular list containing all messages currently in the queue.

Each message is broken into one or more pages, which are dynamically allocated. The beginning of the first page stores the message header, which is a data structure of type msg_msg; its fields are listed in Table 19-13. The m_list field stores the pointers

to the previous and next messages in the queue. The message text starts right after the msg_msg descriptor; if the message is longer than 4,072 bytes (the page size minus the size of the msg_msg descriptor), it continues on another page, whose address is stored in the next field of the msg_msg descriptor. The second page frame starts with a descriptor of type msg_msgseg, which simply includes a next pointer storing the address of an optional third page, and so on.

Table 19-13. The msg_msg data structure

Type	Field	Description
struct list_head	m_list	Pointers for message list
long	m_type	Message type
int	m_ts	Message text size
struct msg_msgseg *	next	Next portion of the message
void *	security	Pointer to a security data structure (used by SELinux)

When the message queue is full (either the maximum number of messages or the maximum total size has been reached), processes that try to enqueue new messages may be blocked. The q_senders field of the msg_queue data structure is the head of a list that includes the pointers to the descriptors of all blocked sending processes.

Even receiving processes may be blocked when the message queue is empty (or the process specified a type of message not present in the queue). The q_receivers field of the msg_queue data structure is the head of a list of msg_receiver data structures, one for every blocked receiving process. Each of these structures essentially includes a pointer to the process descriptor, a pointer to the msg_msg structure of the message, and the type of the requested message.

IPC Shared Memory

The most useful IPC mechanism is shared memory, which allows two or more processes to access some common data structures by placing them in an *IPC shared memory region*. Each process that wants to access the data structures included in an IPC shared memory region must add to its address space a new memory region (see the section "Memory Regions" in Chapter 9), which maps the page frames associated with the IPC shared memory region. Such page frames can then be easily handled by the kernel through demand paging (see the section "Demand Paging" in Chapter 9).

As with semaphores and message queues, the shmget( ) function is invoked to get the IPC identifier of a shared memory region, optionally creating it if it does not already exist.

The shmat( ) function is invoked to "attach" an IPC shared memory region to a process. It receives as its parameter the identifier of the IPC shared memory resource and tries to add a shared memory region to the address space of the calling process. The

calling process can require a specific starting linear address for the memory region, but the address is usually unimportant, and each process accessing the shared memory region can use a different address in its own address space. The process's Page Tables are left unchanged by shmat( ). We describe later what the kernel does when the process tries to access a page that belongs to the new memory region.

The shmdt( ) function is invoked to "detach" an IPC shared memory region specified by its IPC identifier—that is, to remove the corresponding memory region from the process's address space. Recall that an IPC shared memory resource is persistent: even if no process is using it, the corresponding pages cannot be discarded, although they can be swapped out.

As for the other types of IPC resources, in order to avoid overuse of memory by User Mode processes, there are some limits on the allowed number of IPC shared memory regions (by default, 4,096), on the size of each segment (by default, 32 megabytes), and on the maximum total size of all segments (by default, 8 gigabytes). As usual, however, the system administrator can tune these values by writing into the /proc/sys/kernel/shmmni, /proc/sys/kernel/shmmax, and /proc/sys/kernel/shmall files, respectively.
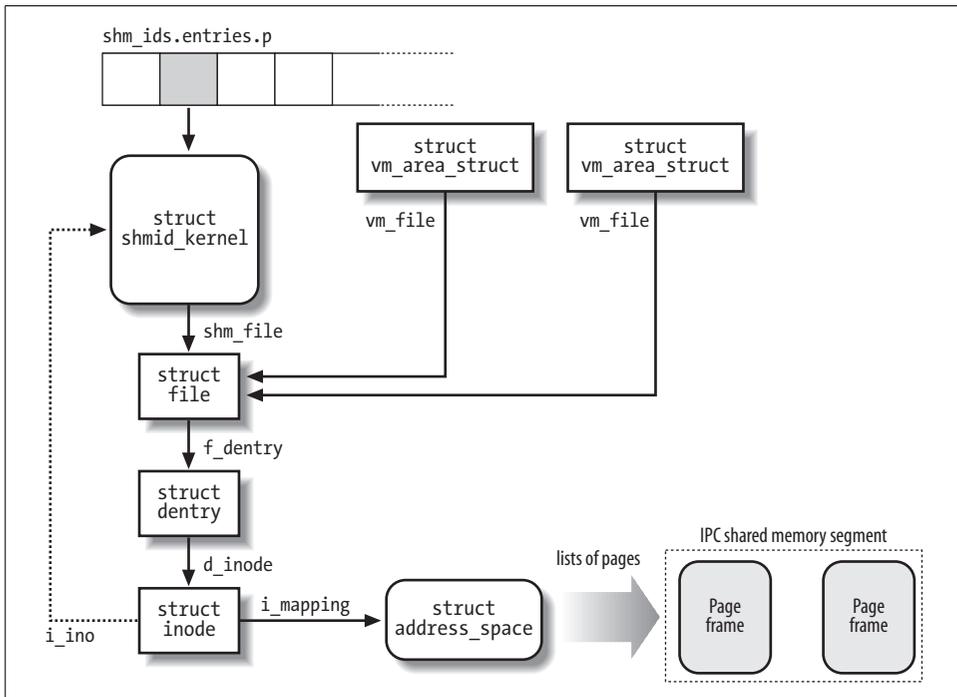
Figure 19-3. IPC shared memory data structures

The data structures associated with IPC shared memory regions are shown in Figure 19-3. The shm_ids variable stores the ipc_ids data structure of the IPC shared

memory resource type; the corresponding `ipc_id_ary` data structure contains an array of pointers to `shmid_kernel` data structures, one item for every IPC shared memory resource. Formally, the array stores pointers to `kern_ipc_perm` data structures, but each such structure is simply the first field of the `msg_queue` data structure. All fields of the `shmid_kernel` data structure are shown in Table 19-14.

Table 19-14. The fields in the shmid_kernel data structure

Type	Field	Description
struct kern_ipc_perm	shm_perm	kern_ipc_perm data structure
struct file *	shm_file	Special file of the segment
int	id	Slot index of the segment
unsigned long	shm_nattch	Number of current attaches
unsigned long	shm_segsz	Segment size in bytes
long	shm_atim	Last access time
long	shm_dtim	Last detach time
long	shm_ctim	Last change time
int	shm_cprid	PID of creator
int	shm_lprid	PID of last accessing process
struct user_struct *	mlock_user	Pointer to the user_struct descriptor of the user that locked in RAM the shared memory resource (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3)

The most important field is `shm_file`, which stores the address of a file object. This reflects the tight integration of IPC shared memory with the VFS layer in Linux 2.6. In particular, each IPC shared memory region is associated with a file belonging to the *shm* special filesystem (see the section "Special Filesystems" in Chapter 12).

Because the *shm* filesystem has no mount point in the system directory tree, no user can open and access its files by means of regular VFS system calls. However, when a process "attaches" a segment, the kernel invokes *do_mmap( )* and creates a new shared memory mapping of the file in the address space of the process. Therefore, files that belong to the *shm* special filesystem have just one file object method, `mmap`, which is implemented by the `shm_mmap( )` function.

As shown in Figure 19-3, a memory region that corresponds to an IPC shared memory region is described by a `vm_area_struct` object (see the section "Memory Mapping" in Chapter 16); its `vm_file` field points back to the file object of the file in the special filesystem, which in turn references a dentry object and an inode object. The inode number, stored in the `i_ino` field of the inode, is actually the slot index of the IPC shared memory region, so the inode object indirectly references the `shmid_kernel` descriptor.

As usual for every shared memory mapping, page frames are included in the page cache through an `address_space` object, which is embedded in the inode and referenced by the `i_mapping` field of the inode (you might also refer to Figure 16-2); in

case of page frames belonging to an IPC shared memory region, the methods of the address_space object are stored in the shmem_aops global variable.

Swapping out pages of IPC shared memory regions

The kernel has to be careful when swapping out pages included in shared memory regions, and the role of the swap cache is crucial (this topic was already discussed in the section "The Swap Cache" in Chapter 17).

Pages of an IPC shared memory region are swappable—and not syncable (see Table 17-1 in Chapter 17)—because they map a special inode that has no image on disk. Thus, in order to reclaim a page of an IPC shared memory region, the kernel must write it into a swap area. Because an IPC shared memory region is persistent— that is, its pages must be preserved even when the segment is not attached to any process—the kernel cannot simply discard these pages even when they are no longer used by any process.

Let us see how the PFRA performs the reclaiming of a page frame used by an IPC shared memory region. Everything is done as described in the section "Low On Memory Reclaiming" in Chapter 17, until the page is considered by shrink_list(). Because this function does not include any special check for pages of IPC shared memory regions, it ends up invoking the try_to_unmap() function to remove every reference to the page frame from the User Mode address spaces; as explained in the section "Reverse Mapping" in Chapter 17, the corresponding page table entries are simply cleared.

Next, the shrink_list() function checks the PG_dirty flag of the page and invokes pageout()—page frames of IPC shared memory regions are marked dirty when they are allocated, thus pageout() is always invoked. In turn, the pageout() function invokes the writepage method of the address_space object of the mapped file.

The shmem_writepage() function, which implements the writepage method for IPC shared memory regions' pages, essentially allocates a new page slot in a swap area, and moves the page from the page cache to the swap cache (it's just a matter of changing the owner address_space object of the page). The function also stores the swapped-out page identifier in a shmem_inode_info structure that embodies the IPC memory region's inode object, and it sets again the PG_dirty flag of the page. As shown in Figure 17-5 in Chapter 17, the shrink_list() function checks the PG_dirty flag and breaks the reclaiming procedure by leaving the page in the inactive list.

Sooner or later, the page frame will be processed again by the PFRA. Once again, the shrink_list() function will try to flush the page to disk by invoking pageout(). This time, however, the page is included in the swap cache, thus it is "owned" by the address_space object of the swapping subsystem, swapper_space. The corresponding writepage method, swap_writepage(), effectively starts the write operation into the swap area (see the section "Swapping Out Pages" in Chapter 17). Once pageout()

terminates, `shrink_list()` verifies that the page is now clean, removes it from the swap cache, and releases it to the buddy system.

Demand paging for IPC shared memory regions

The pages added to a process by `shmat()` are dummy pages; the function adds a new memory region into a process's address space, but it doesn't modify the process's Page Tables. Moreover, as we have seen, pages of an IPC shared memory region can be swapped out. Therefore, these pages are handled through the demand paging mechanism.

As we know, a Page Fault occurs when a process tries to access a location of an IPC shared memory region whose underlying page frame has not been assigned. The corresponding exception handler determines that the faulty address is inside the process address space and that the corresponding Page Table entry is null; therefore, it invokes the `do_no_page()` function (see the section "Demand Paging" in Chapter 9). In turn, this function checks whether the `nopage` method for the memory region is defined. That method is invoked, and the Page Table entry is set to the address returned from it (see also the section "Demand Paging for Memory Mapping" in Chapter 16).

Memory regions used for IPC shared memory always define the `nopage` method. It is implemented by the `shmem_nopage()` function, which performs the following operations:

1. Walks the chain of pointers in the VFS objects and derives the address of the inode object of the IPC shared memory resource (see Figure 19-3).

2. Computes the logical page number inside the segment from the `vm_start` field of the memory region descriptor and the requested address.

3. Checks whether the page is already included in the page cache; if so, terminates by returning the address of its descriptor.

4. Checks whether the page is included in the swap cache and is up-to-date; if so, terminates by returning the address of its descriptor.

5. Checks whether the `shmem_inode_info` that embodies the inode object stores a swapped-out page identifier for the logical page number. If so, it performs a swap-in operation by invoking `read_swap_cache_async()` (see the section "Swapping in Pages" in Chapter 17), waits until the data transfer completes, and terminates by returning the address of the page descriptor.

6. Otherwise, the page is not stored in a swap area; therefore, the function allocates a new page from the buddy system, inserts it into the page cache, and returns its address.

The `do_no_page()` function sets the entry that corresponds to the faulty address in the process's Page Table so that it points to the page frame returned by the method.

POSIX Message Queues

The POSIX standard (IEEE Std 1003.1-2001) defines an IPC mechanism based on message queues, which is usually known as *POSIX message queues*. They are much like the System V IPC's message queues already examined in the section "IPC Messages" earlier in this chapter. However, POSIX message queues sport a number of advantages over the older queues:

- A much simpler file-based interface to the applications
- Native support for message priorities (the priority ultimately determines the position of the message in the queue)
- Native support for asynchronous notification of message arrivals, either by means of signals or thread creation
- Timeouts for blocking send and receive operations

POSIX message queues are handled by means of a set of library functions, which are shown in Table 19-15.

Table 19-15. Library functions for POSIX message queues

Function names	Description
`mq_open()`	Open (optionally creating) a POSIX message queue
`mq_close()`	Close a POSIX message queue (without destroying it)
`mq_unlink()`	Destroy a POSIX message queue
`mq_send()`, `mq_timedsend()`	Send a message to a POSIX message queue; the latter function defines a time limit for the operation
`mq_receive()`, `mq_timedreceive()`	Fetch a message from a POSIX message queue; the latter function defines a time limit for the operation
`mq_notify()`	Establish an asynchronous notification mechanism for the arrival of messages in an empty POSIX message queue
`mq_getattr()`, `mq_setattr()`	Respectively get and set attributes of a POSIX message queue (essentially, whether the send and receive operations should be blocking or nonblocking)

Let's see how an application typically makes use of these functions. As a first step, the application invokes the `mq_open()` library function to open a POSIX message queue. The first argument of the function is a string specifying the name of the queue; it is similar to a filename, and indeed it must start with a slash (/). The library function accepts a subset of the flags of the `open()` system call: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_EXCL`, and `O_NONBLOCK` (for nonblocking send and receive operations). Notice that the application may create a new POSIX message queue by specifying the `O_CREAT` flag. The `mq_open()` function returns a descriptor for the queue—much like the file descriptor returned by the `open()` system call.

Once a POSIX message queue has been opened, the application may send and receive messages by using the library functions `mq_send()` and `mq_receive()`, passing

to them the queue descriptor returned by mq_open(). The application may also make use of mq_timedsend() and mq_timedreceive() to specify the maximum time that the application will spend waiting for the send or receive operation to complete.

Rather than blocking in mq_receive()—or continuously polling the message queue if the O_NONBLOCK flag was specified—the application might also establish an asynchronous notification mechanism by executing the mq_notify() library function. Essentially, the application may require that when a message is inserted in an empty queue, either a signal is sent to a selected process, or a new thread is created.

Finally, when the application has finished using the message queue, it invokes the mq_close() library function; passing to it the queue descriptor. Notice that this function does not destroy the queue, exactly as the close() system call does not remove a file. To destroy a queue, the application makes use of the mq_unlink() function.

The implementation of POSIX message queues in Linux 2.6 is simple and straightforward. A special filesystem named *mqueue* (see the section "Special Filesystems" in Chapter 12) has been introduced, which contains an inode for each existing queue. The kernel offers a few system calls, which roughly correspond to the library functions listed in Table 19-15 earlier: mq_open(), mq_unlink(), mq_timedsend(), mq_timedreceive(), mq_notify(), and mq_getsetattr(). These system calls act transparently on the files of the *mqueue* filesystem, thus much of the job is done by the VFS layer. For example, notice that the kernel does not offer a mq_close() function: in fact, the queue descriptor returned to the application is effectively a file descriptor, therefore the mq_close() library function can simply execute the close() system call to do its job.

The *mqueue* special filesystem must not necessarily be mounted over the system directory tree. However, if it is mounted, a user can create a POSIX message queue by touching a file in the root directory of the filesystem; she can also get information about the queue by reading the corresponding file. Finally, an application can use select() and poll() to be notified about changes in the queue state.

Each queue is described by an mqueue_inode_info descriptor, which embodies the inode object associated with the file in the *mqueue* special filesystem. When a POSIX message queue system call receives a queue descriptor as parameter, it invokes the VFS's fget() function to derive the address of the corresponding file object; next, the system call gets the inode object of the file in the *mqueue* filesystem, and finally the address of the mqueue_inode_info descriptor that contains the inode object.

The pending messages in a queue are collected in a singly linked list rooted at the mqueue_inode_info descriptor; each message is represented by a descriptor of type msg_msg—exactly the same descriptor used for the System V IPC's messages described in the section "IPC Messages" earlier in this chapter.

CHAPTER 20
Program Execution

The concept of a "process," described in Chapter 3, was used in Unix from the beginning to represent the behavior of groups of running programs that compete for system resources. This final chapter focuses on the relationship between program and process. We specifically describe how the kernel sets up the execution context for a process according to the contents of the program file. While it may not seem like a big problem to load a bunch of instructions into memory and point the CPU to them, the kernel has to deal with flexibility in several areas:

Different executable formats
> Linux is distinguished by its ability to run binaries that were compiled for other operating systems. In particular, Linux is able to run an executable created for a 32-bit machine on the 64-bit version of the same machine. For instance, an executable created on a Pentium can run on a 64-bit AMD Opteron.

Shared libraries
> Many executable files don't contain all the code required to run the program but expect the kernel to load in functions from a library at runtime.

Other information in the execution context
> This includes the command-line arguments and environment variables familiar to programmers.

A program is stored on disk as an *executable file*, which includes both the object code of the functions to be executed and the data on which these functions will act. Many functions of the program are service routines available to all programmers; their object code is included in special files called "libraries." Actually, the code of a library function may either be statically copied into the executable file (static libraries) or linked to the process at runtime (shared libraries, because their code can be shared by several independent processes).

When launching a program, the user may supply two kinds of information that affect the way it is executed: command-line arguments and environment variables. *Command-line arguments* are typed in by the user following the executable filename at the

shell prompt. *Environment variables*, such as `HOME` and `PATH`, are inherited from the shell, but the users may modify the values of such variables before they launch the program.

In the section "Executable Files," we explain what a program execution context is. In the section "Executable Formats," we mention some of the executable formats supported by Linux and show how Linux can change its "personality" to execute programs compiled for other operating systems. Finally, in the section "The exec Functions," we describe the system call that allows a process to start executing a new program.

Executable Files

Chapter 1 defined a process as an "execution context." By this we mean the collection of information needed to carry on a specific computation; it includes the pages accessed, the open files, the hardware register contents, and so on. An *executable file* is a regular file that describes how to initialize a new execution context (i.e., how to start a new computation).

Suppose a user wants to list the files in the current directory; he knows that this result can be simply achieved by typing the filename of the */bin/ls*[*] external command at the shell prompt. The command shell forks a new process, which in turn invokes an `execve( )` system call (see the section "The exec Functions" later in this chapter), passing as one of its parameters a string that includes the full pathname for the *ls* executable file—*/bin/ls*, in this case. The `sys_execve( )` service routine finds the corresponding file, checks the executable format, and modifies the execution context of the current process according to the information stored in it. As a result, when the system call terminates, the process starts executing the code stored in the executable file, which performs the directory listing.

When a process starts running a new program, its execution context changes drastically because most of the resources obtained during the process's previous computations are discarded. In the preceding example, when the process starts executing */bin/ls*, it replaces the shell's arguments with new ones passed as parameters in the `execve( )` system call and acquires a new shell environment (see the later section "Command-Line Arguments and Shell Environment"). All pages inherited from the parent (and shared with the Copy On Write mechanism) are released so that the new computation starts with a fresh User Mode address space; even the privileges of the process could change (see the later section "Process Credentials and Capabilities"). However, the process PID doesn't change, and the new computation inherits from the previous one all

---

[*] The pathnames of executable files are not fixed in Linux; they depend on the distribution used. Several standard naming schemes, such as *Filesystem Hierarchy Standard* (*FHS*), have been proposed for all Unix systems.

open file descriptors that were not closed automatically while executing the execve( )
system call.*

Process Credentials and Capabilities

Traditionally, Unix systems associate with each process some *credentials*, which bind
the process to a specific user and a specific user group. Credentials are important on
multiuser systems because they determine what each process can or cannot do, thus
preserving both the integrity of each user's personal data and the stability of the sys-
tem as a whole.

The use of credentials requires support both in the process data structure and in the
resources being protected. One obvious resource is a file. Thus, in the Ext2 file-
system, each file is owned by a specific user and is bound to a group of users. The
owner of a file may decide what kind of operations are allowed on that file, distin-
guishing among herself, the file's user group, and all other users. When a process
tries to access a file, the VFS always checks whether the access is legal, according to
the permissions established by the file owner and the process credentials.

The process's credentials are stored in several fields of the process descriptor, listed
in Table 20-1. These fields contain identifiers of users and user groups in the system,
which are usually compared with the corresponding identifiers stored in the inodes
of the files being accessed.

Table 20-1. Traditional process credentials

Name	Description
uid, gid	User and group real identifiers
euid, egid	User and group effective identifiers
fsuid, fsgid	User and group effective identifiers for file access
groups	Supplementary group identifiers
suid, sgid	User and group saved identifiers

A UID of 0 specifies the superuser (root), while a user group ID of 0 specifies the root
group. If a process credential stores a value of 0, the kernel bypasses the permission
checks and allows the privileged process to perform various actions, such as those
referring to system administration or hardware manipulation, that are not possible to
unprivileged processes.

When a process is created, it always inherits the credentials of its parent. However,
these credentials can be modified later, either when the process starts executing a

---

* By default, a file already opened by a process stays open after issuing an execve( )system call. However, the
  file is automatically closed if the process has set the corresponding bit in the close_on_exec field of the files_
  struct structure (see Table 12-7 in Chapter 12); this is done by means of the fcntl( )system call.

new program or when it issues suitable system calls. Usually, the uid, euid, fsuid, and suid fields of a process contain the same value. When the process executes a *setuid program*—that is, an executable file whose *setuid* flag is on—the euid and fsuid fields are set to the identifier of the file's owner. Almost all checks involve one of these two fields: fsuid is used for file-related operations, while euid is used for all other operations. Similar considerations apply to the gid, egid, fsgid, and sgid fields that refer to group identifiers.

As an illustration of how the fsuid field is used, consider the typical situation when a user wants to change his password. All passwords are stored in a common file, but he cannot directly edit this file because it is protected. Therefore, he invokes a system program named */usr/bin/passwd*, which has the *setuid* flag set and whose owner is the superuser. When the process forked by the shell executes such a program, its euid and fsuid fields are set to 0—to the PID of the superuser. Now the process can access the file, because, when the kernel performs the access control, it finds a 0 value in fsuid. Of course, the */usr/bin/passwd* program does not allow the user to do anything but change his own password.

Unix's long history teaches the lesson that *setuid programs*—programs that have the *setuid* flag set—are quite dangerous: malicious users could trigger some programming errors (bugs) in the code to force *setuid* programs to perform operations that were never planned by the program's original designers. In the worst case, the entire system's security can be compromised. To minimize such risks, Linux, like all modern Unix systems, allows processes to acquire *setuid* privileges only when necessary and drop them when they are no longer needed. This feature may turn out to be useful when implementing user applications with several protection levels. The process descriptor includes an suid field, which stores the values of the effective identifiers (euid and fsuid) at the *setuid* program startup. The process can change the effective identifiers by means of the setuid( ), setresuid( ), setfsuid( ), and setreuid( ) system calls.[*]

Table 20-2 shows how these system calls affect the process's credentials. Be warned that if the calling process does not already have superuser privileges—that is, if its euid field is not null—these system calls can be used only to set values already included in the process's credential fields. For instance, an average user process can store the value 500 into its fsuid field by invoking the setfsuid( ) system call, but only if one of the other credential fields already holds the same value.

---

[*] A group's effective credentials can be changed by issuing the corresponding setgid( ), setresgid( ), setfsgid( ), and setregid( ) system calls.

Table 20-2. Semantics of the system calls that set process credentials

Field	setuid (e)		setresuid (u,e,s)	setreuid (u,e)	setfsuid (f)
	euid=0	euid≠0			
uid	Set to e	Unchanged	Set to u	Set to u	Unchanged
euid	Set to e	Set to e	Set to e	Set to e	Unchanged
fsuid	Set to e	Set to e	Set to e	Set to e	Set to f
suid	Set to e	Unchanged	Set to s	Set to e	Unchanged

To understand the sometimes complex relationships among the four user ID fields, consider for a moment the effects of the setuid( ) system call. The actions are different, depending on whether the calling process's euid field is set to 0 (that is, the process has superuser privileges) or to a normal UID.

If the euid field is 0, the system call sets all credential fields of the calling process (uid, euid, fsuid, and suid) to the value of the parameter e. A superuser process can thus drop its privileges and become a process owned by a normal user. This happens, for instance, when a user logs in: the system forks a new process with superuser privileges, but the process drops its privileges by invoking the setuid( ) system call and then starts executing the user's login shell program.

If the euid field is not 0, the setuid( ) system call modifies only the value stored in euid and fsuid, leaving the other two fields unchanged. This behavior of the system call is useful when implementing a *setuid* program that scales up and down the effective process's privileges stored in the euid and fsuid fields.

Process capabilities

The POSIX.1e draft—now withdrawn—introduced another model of process credentials based on the notion of "capabilities." The Linux kernel supports POSIX capabilities, although most Linux distributions do not make use of them.

A *capability* is simply a flag that asserts whether the process is allowed to perform a specific operation or a specific class of operations. This model is different from the traditional "superuser versus normal user" model in which a process can either do everything or do nothing, depending on its effective UID. As illustrated in Table 20-3, several capabilities have been included in the Linux kernel.

Table 20-3. Linux capabilities

Name	Description
CAP_AUDIT_WRITE	Allow to generate audit messages by writing in netlink sockets
CAP_AUDIT_CONTROL	Allow to control kernel auditing activities by means of netlink sockets
CAP_CHOWN	Ignore restrictions on file user and group ownership changes
CAP_DAC_OVERRIDE	Ignore file access permissions
CAP_DAC_READ_SEARCH	Ignore file/directory read and search permissions

Table 20-3. Linux capabilities (continued)

Name	Description
CAP_FOWNER	Generally ignore permission checks on file ownership
CAP_FSETID	Ignore restrictions on setting the *setuid* and *setgid* flags for files
CAP_KILL	Bypass permission checks when generating signals
CAP_LINUX_IMMUTABLE	Allow modification of append-only and immutable Ext2/Ext3 files
CAP_IPC_LOCK	Allow locking of pages and of shared memory segments
CAP_IPC_OWNER	Skip IPC ownership checks
CAP_LEASE	Allow taking of leases on files (see "Linux File Locking" in Chapter 12)
CAP_MKNOD	Allow privileged `mknod( )` operations
CAP_NET_ADMIN	Allow general networking administration
CAP_NET_BIND_SERVICE	Allow binding to TCP/UDP sockets below 1,024
CAP_NET_BROADCAST	Allow broadcasting and multicasting
CAP_NET_RAW	Allow use of RAW and PACKET sockets
CAP_SETGID	Ignore restrictions on group's process credentials manipulations
CAP_SETPCAP	Allow capability manipulations on other processes
CAP_SETUID	Ignore restrictions on user's process credentials manipulations
CAP_SYS_ADMIN	Allow general system administration
CAP_SYS_BOOT	Allow use of `reboot( )`
CAP_SYS_CHROOT	Allow use of `chroot( )`
CAP_SYS_MODULE	Allow inserting and removing of kernel modules
CAP_SYS_NICE	Skip permission checks of the `nice( )` and `setpriority( )` system calls, and allow creation of real-time processes
CAP_SYS_PACCT	Allow configuration of process accounting
CAP_SYS_PTRACE	Allow use of `ptrace( )` on every process
CAP_SYS_RAWIO	Allow access to I/O ports through `ioperm( )` and `iopl( )`
CAP_SYS_RESOURCE	Allow resource limits to be increased
CAP_SYS_TIME	Allow manipulation of system clock and real-time clock
CAP_SYS_TTY_CONFIG	Allow to configure the terminal and to execute the `vhangup( )` system call

The main advantage of capabilities is that, at any time, each program needs a limited number of them. Consequently, even if a malicious user discovers a way to exploit a buggy program, she can illegally perform only a limited set of operations.

Assume, for instance, that a buggy program has only the CAP_SYS_TIME capability. In this case, the malicious user who discovers an exploitation of the bug can succeed only in illegally changing the real-time clock and the system clock. She won't be able to perform any other kind of privileged operations.

Neither the VFS nor the Ext2 filesystem currently supports the capability model, so there is no way to associate an executable file with the set of capabilities that should

be enforced when a process executes that file. Nevertheless, a process can explicitly get and lower its capabilities by using, respectively, the capget( ) and capset( ) system calls. For instance, it is possible to modify the *login* program to retain a subset of the capabilities and drop the others.

The Linux kernel already takes capabilities into account. Let's consider, for instance, the nice( ) system call, which allows users to change the static priority of a process. In the traditional model, only the superuser can raise a priority; the kernel should therefore check whether the euid field in the descriptor of the calling process is set to 0. However, the Linux kernel defines a capability called CAP_SYS_NICE, which corresponds exactly to this kind of operation. The kernel checks the value of this flag by invoking the capable( ) function and passing the CAP_SYS_NICE value to it.

This approach works, thanks to some "compatibility hacks" that have been added to the kernel code: each time a process sets the euid and fsuid fields to 0 (either by invoking one of the system calls listed in Table 20-2 or by executing a *setuid* program owned by the superuser), the kernel sets all process capabilities so that all checks will succeed. When the process resets the euid and fsuid fields to the real UID of the process owner, the kernel checks the keep_capabilities flag in the process descriptor and drops all capabilities of the process if the flag is set. A process can set and reset the keep_capabilities flag by means of the Linux-specific prctl( ) system call.

The Linux Security Modules framework

In Linux 2.6, capabilities are tightly integrated with the *Linux Security Modules* framework (*LSM*). In short, the LSM framework allows developers to define several alternative models for kernel security.

Each security model is implemented by a set of *security hooks*. A security hook is a function that is invoked by the kernel when it is about to perform an important, security-related operation. The hook function determines whether the operation should be carried on or rejected.

The security hooks are stored in a table of type security_operations. The address of the hook table for the security model currently in use is stored in the security_ops variable. By default, the kernel makes use of a minimal security model implemented by the dummy_security_ops table; each hook in this table essentially checks the corresponding capability, if any, or unconditionally returns 0 (operation allowed).

For instance, the service routines of the stime( ) and settimeofday( ) functions invoke the settime security hook before changing the system date and time. The corresponding function pointed to by the dummy_security_ops table limits itself in checking whether the CAP_SYS_TIME capability of the current process is set, and returns either 0 or -EPERM accordingly.

Sophisticated security models for the Linux kernel have been devised. A widely known example is Security-Enhanced Linux (SELinux), developed by the United State's National Security Agency.

Command-Line Arguments and Shell Environment

When a user types a command, the program that is loaded to satisfy the request may receive some *command-line arguments* from the shell. For example, when a user types the command:

```
$ ls -l /usr/bin
```

to get a full listing of the files in the */usr/bin* directory, the shell process creates a new process to execute the command. This new process loads the */bin/ls* executable file. In doing so, most of the execution context inherited from the shell is lost, but the three separate arguments ls, -l, and /usr/bin are kept. Generally, the new process may receive any number of arguments.

The conventions for passing the command-line arguments depend on the high-level language used. In the C language, the main( ) function of a program may receive as its parameters an integer specifying how many arguments have been passed to the program and the address of an array of pointers to strings. The following prototype formalizes this standard:

```
int main(int argc, char *argv[])
```

Going back to the previous example, when the */bin/ls* program is invoked, argc has the value 3, argv[0] points to the ls string, argv[1] points to the -l string, and argv[2] points to the /usr/bin string. The end of the argv array is always marked by a null pointer, so argv[3] contains NULL.

A third optional parameter that may be passed in the C language to the main( ) function is the parameter containing *environment variables*. They are used to customize the execution context of a process, to provide general information to a user or other processes, or to allow a process to keep some information across an execve( ) system call.

To use the environment variables, main( ) can be declared as follows:

```
int main(int argc, char *argv[], char *envp[])
```

The envp parameter points to an array of pointers to environment strings of the form:

```
VAR_NAME=something
```

where VAR_NAME represents the name of an environment variable, while the substring following the = delimiter represents the actual value assigned to the variable. The end of the envp array is marked by a null pointer, like the argv array. The address of the envp array is also stored in the environ global variable of the C library.

Command-line arguments and environment strings are placed on the User Mode stack, right before the return address (see the section "Parameter Passing" in

Chapter 10). The bottom locations of the User Mode stack are illustrated in Figure 20-1. Notice that the environment variables are located near the bottom of the stack, right after a 0 long integer.
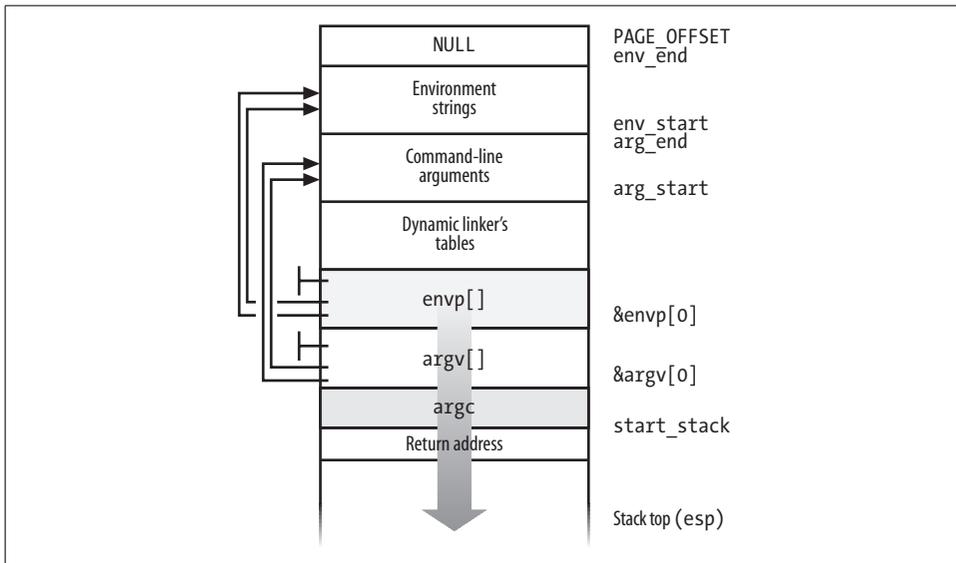
Figure 20-1. The bottom locations of the User Mode stack

Libraries

Each high-level source code file is transformed through several steps into an *object file*, which contains the machine code of the assembly language instructions corresponding to the high-level instructions. An object file cannot be executed, because it does not contain the linear address that corresponds to each reference to a name of a global symbol external to the source code file, such as functions in libraries or other source code files of the same program. The assigning, or *resolution*, of such addresses is performed by the linker, which collects all the object files of the program and constructs the executable file. The linker also analyzes the library's functions used by the program and glues them into the executable file in a manner described later in this chapter.

Most programs, even the most trivial ones, use libraries. Consider, for instance, the following one-line C program:

```
void main(void) { }
```

Although this program does not compute anything, a lot of work is needed to set up the execution environment (see the section "The exec Functions" later in this chapter) and to kill the process when the program terminates (see the section "Destroying Processes" in Chapter 3). In particular, when the main( ) function terminates, the C compiler inserts an exit_group( ) function call in the object code.

We know from Chapter 10 that programs usually invoke system calls through wrapper routines in the C library. This holds for the C compiler, too. Besides including the code directly generated by compiling the program's statements, each executable file also includes some "glue" code to handle the interactions of the User Mode process with the kernel. Portions of such glue code are stored in the C library.

Many other libraries of functions, besides the C library, are included in Unix systems. A generic Linux system typically uses several hundreds of libraries. Just to mention a couple of them: the math library *libm* includes advanced functions for floating point operations, while the X11 library *libX11* collects together the basic low-level functions for the X11 Window System graphics interface.

All executable files in traditional Unix systems were based on *static libraries*. This means that the executable file produced by the linker includes not only the code of the original program but also the code of the library functions that the program refers to. One big disadvantage of statically linked programs is that they eat lots of space on disk. Indeed, each statically linked executable file duplicates some portion of library code.

Modern Unix systems use *shared libraries*. The executable file does not contain the library object code, but only a reference to the library name. When the program is loaded in memory for execution, a suitable program called *dynamic linker* (also named *ld.so*) takes care of analyzing the library names in the executable file, locating the library in the system's directory tree and making the requested code available to the executing process. A process can also load additional shared libraries at runtime by using the dlopen( ) library function.

Shared libraries are especially convenient on systems that provide file memory mapping, because they reduce the amount of main memory requested for executing a program. When the dynamic linker must link a shared library to a process, it does not copy the object code, but performs only a memory mapping of the relevant portion of the library file into the process's address space. This allows the page frames containing the machine code of the library to be shared among all processes that are using the same code. Clearly, sharing is not possible if the program has been linked statically.

Shared libraries also have some disadvantages. The startup time of a dynamically linked program is usually longer than that of a statically linked one. Moreover, dynamically linked programs are not as portable as statically linked ones, because they may not execute properly in systems that include a different version of the same library.

A user may always require a program to be linked statically. For example, the GCC compiler offers the -static option, which tells the linker to use the static libraries instead of the shared ones.

Program Segments and Process Memory Regions

The linear address space of a Unix program is traditionally partitioned, from a logical point of view, in several linear address intervals called segments:[*]

Text segment

Includes the program's executable code.

Initialized data segment

Contains the initialized data—that is, the static variables and the global variables whose initial values are stored in the executable file (because the program must know their values at startup).

Uninitialized data segment (bss)

Contains the uninitialized data—that is, all global variables whose initial values are not stored in the executable file (because the program sets the values before referencing them); it is historically called a *bss segment*.

Stack segment

Contains the program stack, which includes the return addresses, parameters, and local variables of the functions being executed.

Each `mm_struct` memory descriptor (see the section "The Memory Descriptor" in Chapter 9) includes some fields that identify the role of a few crucial memory regions of the corresponding process:

`start_code, end_code`

Store the initial and final linear addresses of the memory region that includes the native code of the program—the code in the executable file.

`start_data, end_data`

Store the initial and final linear addresses of the memory region that includes the native initialized data of the program, as specified in the executable file. The fields identify a memory region that roughly corresponds to the data segment.

`start_brk, brk`

Store the initial and final linear addresses of the memory region that includes the dynamically allocated memory areas of the process (see the section "Managing the Heap" in Chapter 9). This memory region is sometimes called the *heap*.

`start_stack`

Stores the address right above that of `main( )`'s return address; as illustrated in Figure 20-1, higher addresses are reserved (recall that stacks grow toward lower addresses).

---

[*] The word "segment" has historical roots, because the first Unix systems implemented each linear address interval with a different segment register. Linux, however, does not rely on the segmentation mechanism of the 80x86 microprocessors to implement program segments.

```
arg_start, arg_end
```
   Store the initial and final addresses of the stack portion containing the com-
   mand-line arguments.

```
env_start, env_end
```
   Store the initial and final addresses of the stack portion containing the environ-
   ment strings.

Notice that shared libraries and file memory mapping have made the classification of
the process's address space based on program segments obsolete, because each of the
shared libraries is mapped into a different memory region from those discussed in
the preceding list.

Flexible memory region layout

The *flexible memory region layout* has been introduced in the kernel version 2.6.9:
essentially, each process gets a memory layout that depends on how much the User
Mode stack is expected to grow. However, the old, classical layout can still be used
(mainly when the kernel cannot put a limit on the size of the User Mode stack of a
process). Both layouts are described in Table 20-4, assuming the 80*x*86 architecture
with the default User Mode address space spanning up to 3 GB.

*Table 20-4. The memory region layouts in the 80*x*86 architecture*

Type of memory region	Classical layout	Flexible layout
Text segment (ELF)	Starts from 0x08048000	
Data and bss segments	Starts right after the text segment	
Heap	Starts right after the data and bss segments	
File memory mappings and anonymous memory regions	Starts from 0x40000000 (this address corresponds to 1/3 of the whole User Mode address space); libraries added at successively higher addresses	Starts near the end (lowest address) of the User Mode stack; libraries added at successively lower addresses
User Mode stack	Starts at 0xc0000000 and grows towards lower addresses	

As you can see, the layouts differ only on the position of the memory regions for file
memory mappings and anonymous mappings. In the classical layout, these regions
are placed starting at one-third of the whole User Mode address space, usually at
0x40000000; newer regions are added at higher linear addresses, thus the regions
expand towards the User Mode stack.

Conversely, in the flexible layout the memory regions for file memory mapping and
anonymous mappings are placed near the end of the User Mode stack; newer regions
are added at lower linear addresses, thus the regions expand towards the heap.
Remember that the stack grows towards lower addresses, too.

The kernel typically uses the flexible layout when it can get a limit on the size of
the User Mode stack by means of the RLIMIT_STACK resource limit (see the section

"Process Resource Limits" in Chapter 3). This limit determines the size of the linear address space reserved for the stack; however, this size cannot be smaller than 128 MB or larger than 2.5 GB.

On the other hand, if either the RLIMIT_STACK resource limit is set to "infinity" or the system administrator has set to 1 the sysctl_legacy_va_layout variable (by writing in the /proc/sys/vm/legacy_va_layout file or by issuing the proper sysctl() system call), the kernel cannot determine an upper bound on the size of the User Mode stack, thus it sticks to the classical memory region layout.

Why has the flexible layout been introduced? Its main advantage is that it allows a process to make better use of the User Mode linear address space. In the classical layout the heap is limited to less than 1 GB, while the other memory regions can fill up to about 2 GB (minus the stack size). In the flexible layout, these constraints are gone: both the heap and the other memory regions can freely expand until all the linear addresses left unused by the User Mode stack and the program's fixed-size segments are taken.

At this point, a small, practical experiment can be quite enlightening. Let's write and compile the following C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    char cmd[32];
    brk((void *)0x8051000);
    sprintf(cmd, "cat /proc/self/maps");
    system(cmd);
    return 0;
}
```

Essentially, the program enlarges the heap of the process (see the section "Managing the Heap" in Chapter 9), then it reads the *maps* file in the */proc* special filesystem that produces the list of memory regions of the process itself.

Let's run the program without putting any limit on the stack size:

```
# ulimit -s unlimited; /tmp/memorylayout
08048000-08049000 r-xp 00000000 03:03 5042408    /tmp/memorylayout
08049000-0804a000 rwxp 00000000 03:03 5042408    /tmp/memorylayout
0804a000-08051000 rwxp 0804a000 00:00 0
40000000-40014000 r-xp 00000000 03:03 620801     /lib/ld-2.3.2.so
40014000-40015000 rwxp 00013000 03:03 620801     /lib/ld-2.3.2.so
40015000-40016000 rwxp 40015000 00:00 0
4002f000-40157000 r-xp 00000000 03:03 620804     /lib/libc-2.3.2.so
40157000-4015b000 rwxp 00128000 03:03 620804     /lib/libc-2.3.2.so
4015b000-4015e000 rwxp 4015b000 00:00 0
bffeb000-c0000000 rwxp bffeb000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

(You might see a slightly different table, depending on the version of the C compiler suite and on how the program has been linked.) The first two hexadecimal numbers represent the extent of the memory region; they are followed by the permission flags; finally, there is some information about the file mapped by the memory region, if any: the starting offset inside the file, the block device number and the inode number, and the filename.

Notice that all regions listed are implemented by means of private memory mappings (the letter p in the permission column). This is not surprising because these memory regions exist only to provide data to a process. While executing instructions, a process may modify the contents of these memory regions; however, the files on disk associated with them stay unchanged. This is precisely how private memory mappings act.

The memory region starting from 0x8048000 is a memory mapping associated with the portion of the */tmp/memorylayout* file ranging from byte 0 to byte 4,095. The permissions specify that the region is executable (it contains object code), read-only (it's not writable because the instructions don't change during a run), and private. That's correct, because the region maps the text segment of the program.

The memory region starting from 0x8049000 is another memory mapping associated with the same portion of */tmp/memorylayout* ranging from byte 0 to byte 4,095. This program is so small that the text, data, and bss segments of the program are included in the same file's page. Thus, the memory region containing the data and bss segments overlaps with the previous memory region in the linear address space.

The third memory region contains the heap of the process. Notice that it terminates at the linear address 0x8051000 that was passed to the brk() system call.

The next two memory regions starting from 0x40000000 and 0x40014000 correspond to the text segment and to the data and bss segments, respectively, of the dynamic linker for the ELF shared libraries—*/lib/ld-2.3.2.so* on this system. The dynamic linker is never executed alone: it is always memory-mapped inside the address space of a process executing another program. The anonymous memory region starting from 0x40015000 has been allocated by the dynamic linker.

On this system, the C library happens to be stored in the */lib/libc-2.3.2.so* file. The text segment and the data and bss segments of the C library are mapped into the next two memory regions, starting from address 0x4002f000. Remember that page frames included in private regions can be shared among several processes with the Copy On Write mechanism, as long as they are not modified. Thus, because the text segment is read-only, the page frames containing the executable code of the C library are shared among almost all currently executing processes (all except the statically linked ones). The anonymous memory region starting from 0x4015b000 has been allocated by the C library.

The anonymous memory region from 0xbffeb000 to 0xc0000000 is associated with the User Mode stack. We already explained in the section "Page Fault Exception Handler" in Chapter 9 how the stack is automatically expanded toward lower addresses whenever necessary.

Finally, the one-page anonymous memory region from 0xffffe000 contains the vsyscall page of the process, which is accessed when issuing a system call and returning from a signal handler (see the section "Issuing a System Call via the sysenter Instruction" in Chapter 10 and the section "Catching the Signal" in Chapter 11).

Now let's run the same program by enforcing a limit on the size of the User Mode stack:

```
# ulimit -s 100; /tmp/memorylayout
08048000-08049000 r-xp 00000000 03:03 5042408    /tmp/memorylayout
08049000-0804a000 rwxp 00000000 03:03 5042408    /tmp/memorylayout
0804a000-08051000 rwxp 0804a000 00:00 0
b7ea3000-b7fcb000 r-xp 00000000 03:03 620804     /lib/libc-2.3.2.so
b7fcb000-b7fcf000 rwxp 00128000 03:03 620804     /lib/libc-2.3.2.so
b7fcf000-b7fd2000 rwxp b7fcf000 00:00 0
b7feb000-b7fec000 rwxp b7feb000 00:00 0
b7fec000-b8000000 r-xp 00000000 03:03 620801     /lib/ld-2.3.2.so
b8000000-b8001000 rwxp 00013000 03:03 620801     /lib/ld-2.3.2.so
bffeb000-c0000000 rwxp bffeb000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

Notice how the layout has changed: the dynamic linker has been mapped about 128 MB above the highest stack address. Furthermore, because the memory regions of the C library have been created later, they get lower linear addresses.

Execution Tracing

Execution tracing is a technique that allows a program to monitor the execution of another program. The traced program can be executed step by step, until a signal is received, or until a system call is invoked. Execution tracing is widely used by debuggers, together with other techniques such as the insertion of breakpoints in the debugged program and runtime access to its variables. We focus on how the kernel supports execution tracing rather than discussing how debuggers work.

In Linux, execution tracing is performed through the ptrace() system call, which can handle the commands listed in Table 20-5. Processes having the CAP_SYS_PTRACE capability flag set are allowed to trace every process in the system except *init*. Conversely, a process *P* with no CAP_SYS_PTRACE capability is allowed to trace only processes having the same owner as *P*. Moreover, a process cannot be traced by two processes at the same time.

Table 20-5. The ptrace commands in the 80×86 architecture

Command	Description
PTRACE_ATTACH	Start execution tracing for another process
PTRACE_CONT	Resume execution
PTRACE_DETACH	Terminate execution tracing
PTRACE_GET_THREAD_AREA	Get the Thread Local Storage (TLS) area on behalf of the traced process
PTRACE_GETEVENTMSG	Get additional data from the traced process (e.g., the PID of a newly forked process)
PTRACE_GETFPREGS	Read floating point registers
PTRACE_GETFPXREGS	Read MMX and XMM registers
PTRACE_GETREGS	Read privileged CPU's registers
PTRACE_GETSIGINFO	Get information on the last signal delivered to the traced process
PTRACE_KILL	Kill the traced process
PTRACE_OLDSETOPTIONS	Architecture-dependent command equivalent to PTRACE_SETOPTIONS
PTRACE_PEEKDATA	Read a 32-bit value from the data segment
PTRACE_PEEKTEXT	Read a 32-bit value from the text segment
PTRACE_PEEKUSR	Read the CPU's normal and debug registers
PTRACE_POKEDATA	Write a 32-bit value into the data segment
PTRACE_POKETEXT	Write a 32-bit value into the text segment
PTRACE_POKEUSR	Write the CPU's normal and debug registers
PTRACE_SET_THREAD_AREA	Set the Thread Local Storage (TLS) area on behalf of the traced process
PTRACE_SETFPREGS	Write floating point registers
PTRACE_SETFPXREGS	Write MMX and XMM registers
PTRACE_SETOPTIONS	Modify ptrace( ) behavior
PTRACE_SETREGS	Write privileged CPU's registers
PTRACE_SETSIGINFO	Forge the information on the last signal delivered to the traced process
PTRACE_SINGLESTEP	Resume execution for a single assembly language instruction
PTRACE_SYSCALL	Resume execution until the next system call boundary
PTRACE_TRACEME	Start execution tracing for the current process

The ptrace( ) system call modifies the parent field in the descriptor of the traced process so that it points to the tracing process; therefore, the tracing process becomes the effective parent of the traced one. When execution tracing terminates—i.e., when ptrace( ) is invoked with the PTRACE_DETACH command—the system call sets p_pptr to the value of real_parent, thus restoring the original parent of the traced process (see the section "Relationships Among Processes" in Chapter 3).

Several monitored events can be associated with a traced program:

- End of execution of a single assembly language instruction
- Entering a system call

- Exiting from a system call
- Receiving a signal

When a monitored event occurs, the traced program is stopped and a `SIGCHLD` signal is sent to its parent. When the parent wishes to resume the child's execution, it can use one of the `PTRACE_CONT`, `PTRACE_SINGLESTEP`, and `PTRACE_SYSCALL` commands, depending on the kind of event it wants to monitor.

The `PTRACE_CONT` command simply resumes execution; the child executes until it receives another signal. This kind of tracing is implemented by means of the `PT_PTRACED` flag in the `ptrace` field of the process descriptor, which is checked by the `do_signal()` function (see the section "Delivering a Signal" in Chapter 11).

The `PTRACE_SINGLESTEP` command forces the child process to execute the next assembly language instruction, and then stops it again. This kind of tracing is implemented on 80**x**86-based machines by means of the `TF` trap flag in the `eflags` register: when it is on, a "Debug" exception is raised right after every assembly language instruction. The corresponding exception handler just clears the flag, forces the current process to stop, and sends a `SIGCHLD` signal to its parent. Notice that setting the `TF` flag is not a privileged operation, so User Mode processes can force single-step execution even without the `ptrace()` system call. The kernel checks the `PT_DTRACE` flag in the process descriptor to keep track of whether the child process is being single-stepped through `ptrace()`.

The `PTRACE_SYSCALL` command causes the traced process to resume execution until a system call is invoked. The process is stopped twice: the first time when the system call starts and the second time when the system call terminates. This kind of tracing is implemented by means of the `TIF_SYSCALL_TRACE` flag included in the `flags` field of the `thread_info` structure of the process, which is checked in the `system_call()` assembly language function (see the section "Issuing a System Call via the int $0x80 Instruction" in Chapter 10).

A process can also be traced using some debugging features of the Intel Pentium processors. For example, the parent could set the values of the `dr0`, …, `dr7` debug registers for the child by using the `PTRACE_POKEUSR` command. When an event monitored by a debug register occurs, the CPU raises the "Debug" exception; the exception handler can then suspend the traced process and send the `SIGCHLD` signal to the parent.

Executable Formats

The standard Linux executable format is named *Executable and Linking Format (ELF)*. It was developed by Unix System Laboratories and is now the most widely used format in the Unix world. Several well-known Unix operating systems, such as System V Release 4 and Sun's Solaris 2, have adopted ELF as their main executable format.

Older Linux versions supported another format named *Assembler OUTput Format* (*a.out*); actually, there were several versions of that format floating around the Unix world. It is seldom used now, because ELF is much more practical.

Linux supports many other different formats for executable files; in this way, it can run programs compiled for other operating systems, such as MS-DOS EXE programs or BSD Unix's COFF executables. A few executable formats, such as Java or *bash* scripts, are platform-independent.

An executable format is described by an object of type `linux_binfmt`, which essentially provides three methods:

load_binary
> Sets up a new execution environment for the current process by reading the information stored in an executable file.

load_shlib
> Dynamically binds a shared library to an already running process; it is activated by the `uselib( )` system call.

core_dump
> Stores the execution context of the current process in a file named `core`. This file, whose format depends on the type of executable of the program being executed, is usually created when a process receives a signal whose default action is "dump" (see the section "Actions Performed upon Delivering a Signal" in Chapter 11).

All `linux_binfmt` objects are included in a singly linked list, and the address of the first element is stored in the `formats` variable. Elements can be inserted and removed in the list by invoking the `register_binfmt( )` and `unregister_binfmt( )` functions. The `register_binfmt( )` function is executed during system startup for each executable format compiled into the kernel. This function is also executed when a module implementing a new executable format is being loaded, while the `unregister_binfmt( )` function is invoked when the module is unloaded.

The last element in the `formats` list is always an object describing the executable format for *interpreted scripts*. This format defines only the `load_binary` method. The corresponding `load_script( )` function checks whether the executable file starts with the #! pair of characters. If so, it interprets the rest of the first line as the pathname of another executable file and tries to execute it by passing the name of the script file as a parameter.[*]

---

[*] It is possible to execute a script file even if it doesn't start with the #! characters, as long as the file is written in the language recognized by a command shell. In this case, however, the script is interpreted either by the shell on which the user types the command or by the default Bourne shell *sh*; therefore, the kernel is not directly involved.

Linux allows users to register their own custom executable formats. Each such format may be recognized either by means of a magic number stored in the first 128 bytes of the file, or by a filename extension that identifies the file type. For example, MS-DOS extensions consist of three characters separated from the filename by a dot: the *.exe* extension identifies executable programs, while the *.bat* extension identifies shell scripts.

When the kernel determines that the executable file has a custom format, it starts the proper interpreter program. The *interpreter program* runs in User Mode, receives as its parameter the pathname of the executable file, and carries on the computation. As an example, an executable file containing a Java program is dealt by a java virtual machine such as */usr/lib/java/bin/java*.

The mechanism is similar to the script's format, but it's more powerful because it doesn't impose any restrictions on the custom format. To register a new format, the user writes into the *register* file of the *binfmt_misc* special filesystem (usually mounted on */proc/sys/fs/binfmt_misc*) a string with the following format:

```
:name:type:offset:string:mask:interpreter:flags
```

where each field has the following meaning:

name
    An identifier for the new format

type
    The type of recognition (M for magic number, E for extension)

offset
    The starting offset of the magic number inside the file

string
    The byte sequence to be matched either in the magic number or in the extension

mask
    The string to mask out some bits in string

interpreter
    The full pathname of the interpreter program

flags
    Some optional flags that control how the interpreter program has to be invoked

For example, the following command performed by the superuser enables the kernel to recognize the Microsoft Windows executable format:

```
$ echo ':DOSWin:M:0:MZ:0xff:/usr/bin/wine:'
                               > /proc/sys/fs/binfmt_misc/register
```

A Windows executable file has the MZ magic number in the first two bytes, and it is executed by the */usr/bin/wine* interpreter program.

Execution Domains

As mentioned in Chapter 1, a neat feature of Linux is its ability to execute files compiled for other operating systems. Of course, this is possible only if the files include machine code for the same computer architecture on which the kernel is running. Two kinds of support are offered for these "foreign" programs:

- Emulated execution: necessary to execute programs that include system calls that are not POSIX-compliant
- Native execution: valid for programs whose system calls are totally POSIX-compliant

Microsoft MS-DOS and Windows programs are emulated: they cannot be natively executed, because they include APIs that are not recognized by Linux. An emulator such as DOSemu or Wine (which appeared in the example at the end of the previous section) is invoked to translate each API call into an emulating wrapper function call, which in turn uses the existing Linux system calls. Because emulators are mostly implemented as User Mode applications, we don't discuss them further.

On the other hand, POSIX-compliant programs compiled on operating systems other than Linux can be executed without too much trouble, because POSIX operating systems offer similar APIs. (Actually, the APIs should be identical, although this is not always the case.) Minor differences that the kernel must iron out usually refer to how system calls are invoked or how the various signals are numbered. This information is stored in *execution domain descriptors* of type exec_domain.

A process specifies its execution domain by setting the personality field of its descriptor and storing the address of the corresponding exec_domain data structure in the exec_domain field of the thread_info structure. A process can change its personality by issuing a suitable system call named personality( ); typical values assumed by the system call's parameter are listed in Table 20-6. Programmers are not expected to directly change the personality of their programs; instead, the personality( ) system call should be issued by the glue code that sets up the execution context of the process (see the next section).

Table 20-6. Personalities supported by the Linux kernel

Personality	Operating system
PER_LINUX	Standard execution domain
PER_LINUX_32BIT	Linux with 32-bit physical addresses in 64-bit architectures
PER_LINUX_FDPIC	Linux program in ELF FDPIC format
PER_SVR4	System V Release 4
PER_SVR3	System V Release 3
PER_SCOSVR3	SCO Unix Version 3.2
PER_OSR5	SCO OpenServer Release 5

Table 20-6. Personalities supported by the Linux kernel (continued)

Personality	Operating system
PER_WYSEV386	Unix System V/386 Release 3.2.1
PER_ISCR4	Interactive Unix
PER_BSD	BSD Unix
PER_SUNOS	SunOS
PER_XENIX	Xenix
PER_LINUX32	Emulation of Linux 32-bit programs in 64-bit architectures (using a 4 GB User Mode address space)
PER_LINUX32_3GB	Emulation of Linux 32-bit programs in 64-bit architectures (using a 3 GB User Mode address space)
PER_IRIX32	SGI IRIX-5 32 bit
PER_IRIXN32	SGI IRIX-6 32 bit
PER_IRIX64	SGI IRIX-6 64 bit
PER_RISCOS	RISC OS
PER_SOLARIS	Sun's Solaris
PER_UW7	SCO's (formerly Caldera's) UnixWare 7
PER_OSF4	Digital UNIX (Compaq Tru64 UNIX)
PER_HPUX	Hewlett-Packard's HP-UX

The exec Functions

Unix systems provide a family of functions that replace the execution context of a process with a new context described by an executable file. The names of these functions start with the prefix exec, followed by one or two letters; therefore, a generic function in the family is usually referred to as an exec function.

The exec functions are listed in Table 20-7; they differ in how the parameters are interpreted.

Table 20-7. The exec functions

Function name	PATH search	Command-line arguments	Environment array
execl( )	No	List	No
execlp( )	Yes	List	No
execle( )	No	List	Yes
execv( )	No	Array	No
execvp( )	Yes	Array	No
execve( )	No	Array	Yes

The first parameter of each function denotes the pathname of the file to be executed. The pathname can be absolute or relative to the process's current directory.

Moreover, if the name does not include any / characters, the execlp( ) and execvp( ) functions search for the executable file in all directories specified by the PATH environment variable.

Besides the first parameter, the execl( ), execlp( ), and execle( ) functions include a variable number of additional parameters. Each points to a string describing a command-line argument for the new program; as the "l" character in the function names suggests, the parameters are organized in a list terminated by a NULL value. Usually, the first command-line argument duplicates the executable filename. Conversely, the execv( ), execvp( ), and execve( ) functions specify the command-line arguments with a single parameter; as the v character in the function names suggests, the parameter is the address of a vector of pointers to command-line argument strings. The last component of the array must be NULL.

The execle( ) and execve( ) functions receive as their last parameter the address of an array of pointers to environment strings; as usual, the last component of the array must be NULL. The other functions may access the environment for the new program from the external environ global variable, which is defined in the C library.

All exec functions, with the exception of execve( ), are wrapper routines defined in the C library and use execve( ), which is the only system call offered by Linux to deal with program execution.

The sys_execve( ) service routine receives the following parameters:

- The address of the executable file pathname (in the User Mode address space).
- The address of a NULL-terminated array (in the User Mode address space) of pointers to strings (again in the User Mode address space); each string represents a command-line argument.
- The address of a NULL-terminated array (in the User Mode address space) of pointers to strings (again in the User Mode address space); each string represents an environment variable in the NAME=value format.

The function copies the executable file pathname into a newly allocated page frame. It then invokes the do_execve( ) function, passing to it the pointers to the page frame, to the pointer's arrays, and to the location of the Kernel Mode stack where the User Mode register contents are saved. In turn, do_execve( ) performs the following operations:

1. Dynamically allocates a linux_binprm data structure, which will be filled with data concerning the new executable file.
2. Invokes path_lookup( ), dentry_open( ), and path_release( ) to get the dentry object, the file object, and the inode object associated with the executable file. On failure, it returns the proper error code.
3. Verifies that the file is executable by the current process; also, checks that the file is not being written by looking at the i_writecount field of the inode; stores –1 in that field to forbid further write accesses.

4. In multiprocessor systems, it invokes the `sched_exec()` function to determine the least loaded CPU that can execute the new program and to migrate the current process to it (see Chapter 7).

5. Invokes `init_new_context()` to check whether the current process was using a custom Local Descriptor Table (see the section "The Linux LDTs" in Chapter 2); in this case, the function allocates and fills a new LDT to be used by the new program.

6. Invokes the `prepare_binprm()` function to fill the `linux_binprm` data structure. This function, in turn, performs the following operations:

    a. Checks again whether the file is executable (at least one execute access right is set); if not, returns an error code. (The previous check in step 3 is not sufficient because a process with the `CAP_DAC_OVERRIDE` capability set always satisfies the check; see the section "Process Credentials and Capabilities" earlier in this chapter).

    b. Initializes the `e_uid` and `e_gid` fields of the `linux_binprm` structure, taking into account the values of the *setuid* and *setgid* flags of the executable file. These fields represent the effective user and group IDs, respectively. Also checks process capabilities (a compatibility hack explained in the earlier section "Process Credentials and Capabilities").

    c. Fills the `buf` field of the `linux_binprm` structure with the first 128 bytes of the executable file. These bytes include the magic number of the executable format and other information suitable for recognizing the executable file.

7. Copies the file pathname, command-line arguments, and environment strings into one or more newly allocated page frames. (Eventually, they are assigned to the User Mode address space.)

8. Invokes the `search_binary_handler()` function, which scans the `formats` list and tries to apply the `load_binary` method of each element, passing to it the `linux_binprm` data structure. The scan of the `formats` list terminates as soon as a `load_binary` method succeeds in acknowledging the executable format of the file.

9. If the executable file format is not present in the `formats` list, it releases all allocated page frames and returns the error code `-ENOEXEC`. Linux cannot recognize the executable file format.

10. Otherwise, the function releases the `linux_binprm` data structure and returns the code obtained from the `load_binary` method associated with the executable format of the file.

The `load_binary` method corresponding to an executable file format performs the following operations (we assume that the executable file is stored on a filesystem that allows file memory mapping and that it requires one or more shared libraries):

1. Checks some magic numbers stored in the first 128 bytes of the file to identify the executable format. If the magic numbers don't match, it returns the error code `-ENOEXEC`.

2. Reads the header of the executable file. This header describes the program's segments and the shared libraries requested.

3. Gets from the executable file the pathname of the dynamic linker, which is used to locate the shared libraries and map them into memory.

4. Gets the dentry object (as well as the inode object and the file object) of the dynamic linker.

5. Checks the execution permissions of the dynamic linker.

6. Copies the first 128 bytes of the dynamic linker into a buffer.

7. Performs some consistency checks on the dynamic linker type.

8. Invokes the `flush_old_exec( )` function to release almost all resources used by the previous computation; in turn, this function performs the following operations:

    a. If the table of signal handlers is shared with other processes, it allocates a new table and decrements the usage counter of the old one; moreover, it detaches the process from the old thread group (see the section "Identifying a Process" in Chapter 3). All of this is done by invoking the `de_thread( )` function.

    b. Invokes `unshare_files( )` to make a copy of the `files_struct` structure containing the open files of the process, if it is shared with other processes (see the section "Files Associated with a Process" in Chapter 12).

    c. Invokes the `exec_mmap( )` function to release the memory descriptor, all memory regions, and all page frames assigned to the process and to clean up the process's Page Tables.

    d. Sets the `comm` field of the process descriptor with the executable file pathname.

    e. Invokes the `flush_thread( )` function to clear the values of the floating point registers and debug registers saved in the TSS segment.

    f. Updates the table of signal handlers by resetting each signal to its default action. This is done by invoking the `flush_signal_handlers( )` function.

    g. Invokes the `flush_old_files( )` function to close all open files having the corresponding flag in the `files->close_on_exec` field of the process descriptor set (see the section "Files Associated with a Process" in Chapter 12).[*]

    Now we have reached the point of no return: the function cannot restore the previous computation if something goes wrong.

9. Clears the `PF_FORKNOEXEC` flag in the process descriptor. This flag, which is set when a process is forked and cleared when it executes a new program, is required for process accounting.

---

[*] These flags can be read and modified by means of the `fcntl( )` system call.

10. Sets up the new personality of the process—that is, the `personality` field in the process descriptor.

11. Invokes `arch_pick_mmap_layout()` to select the layout of the memory regions of the process (see the section "Program Segments and Process Memory Regions" earlier in this chapter).

12. Invokes the `setup_arg_pages()` function to allocate a new memory region descriptor for the process's User Mode stack and to insert that memory region into the process's address space. `setup_arg_pages()` also assigns the page frames containing the command-line arguments and the environment variable strings to the new memory region.

13. Invokes the `do_mmap()` function to create a new memory region that maps the text segment (that is, the code) of the executable file. The initial linear address of the memory region depends on the executable format, because the program's executable code is usually not relocatable. Therefore, the function assumes that the text segment is loaded starting from some specific logical address offset (and thus from some specified linear address). ELF programs are loaded starting from linear address `0x08048000`.

14. Invokes the `do_mmap()` function to create a new memory region that maps the data segment of the executable file. Again, the initial linear address of the memory region depends on the executable format, because the executable code expects to find its variables at specified offsets (that is, at specified linear addresses). In an ELF program, the data segment is loaded right after the text segment.

15. Allocates additional memory regions for every other specialized segments of the executable file. Usually, there are none.

16. Invokes a function that loads the dynamic linker. If the dynamic linker is an ELF executable, the function is named `load_elf_interp()`. In general, the function performs the operations in steps 12 through 14, but for the dynamic linker instead of the file to be executed. The initial addresses of the memory regions that will include the text and data of the dynamic linker are specified by the dynamic linker itself; however, they are very high (usually above `0x40000000`) to avoid collisions with the memory regions that map the text and data of the file to be executed (see the earlier section "Program Segments and Process Memory Regions").

17. Stores in the `binfmt` field of the process descriptor the address of the `linux_binfmt` object of the executable format.

18. Determines the new capabilities of the process.

19. Creates specific dynamic linker tables and stores them on the User Mode stack between the command-line arguments and the array of pointers to environment strings (see Figure 20-1).

20. Sets the values of the `start_code`, `end_code`, `start_data`, `end_data`, `start_brk`, `brk`, and `start_stack` fields of the process's memory descriptor.

21. Invokes the `do_brk()` function to create a new anonymous memory region mapping the bss segment of the program. (When the process writes into a variable, it triggers demand paging, and thus the allocation of a page frame.) The size of this memory region was computed when the executable program was linked. The initial linear address of the memory region must be specified, because the program's executable code is usually not relocatable. In an ELF program, the bss segment is loaded right after the data segment.

22. Invokes the `start_thread()` macro to modify the values of the User Mode registers `eip` and `esp` saved on the Kernel Mode stack, so that they point to the entry point of the dynamic linker and to the top of the new User Mode stack, respectively.

23. If the process is being traced, it notifies the debugger about the completion of the `execve()` system call.

24. Returns the value 0 (success).

When the `execve()` system call terminates and the calling process resumes its execution in User Mode, the execution context is dramatically changed: the code that invoked the system call no longer exists. In this sense, we could say that `execve()` never returns on success. Instead, a new program to be executed is mapped in the address space of the process.

However, the new program cannot yet be executed, because the dynamic linker must still take care of loading the shared libraries.[*]

Although the dynamic linker runs in User Mode, we briefly sketch out here how it operates. Its first job is to set up a basic execution context for itself, starting from the information stored by the kernel in the User Mode stack between the array of pointers to environment strings and `arg_start`. Then the dynamic linker must examine the program to be executed to identify which shared libraries must be loaded and which functions in each shared library are effectively requested. Next, the interpreter issues several `mmap()` system calls to create memory regions mapping the pages that will hold the library functions (text and data) actually used by the program. Then the interpreter updates all references to the symbols of the shared library, according to the linear addresses of the library's memory regions. Finally, the dynamic linker terminates its execution by jumping to the main entry point of the program to be executed. From now on, the process will execute the code of the executable file and of the shared libraries.

---

[*] Things are much simpler if the executable file is statically linked—that is, if no shared library is requested. The `load_binary` method simply maps the text, data, bss, and stack segments of the program into the process memory regions, and then sets the User Mode `eip` register to the entry point of the new program.

As you may have noticed, executing a program is a complex activity that involves many facets of kernel design, such as process abstraction, memory management, system calls, and filesystems. It is the kind of topic that makes you realize what a marvelous piece of work Linux is!

System Startup

This appendix explains what happens right after users switch on their computers—that is, how a Linux kernel image is copied into memory and executed. In short, we discuss how the kernel, and thus the whole system, is "bootstrapped."

Traditionally, the term *bootstrap* refers to a person who tries to stand up by pulling his own boots. In operating systems, the term denotes bringing at least a portion of the operating system into main memory and having the processor execute it. It also denotes the initialization of kernel data structures, the creation of some user processes, and the transfer of control to one of them.

Computer bootstrapping is a tedious, long task, because initially, nearly every hardware device, including the RAM, is in a random, unpredictable state. Moreover, the bootstrap process is highly dependent on the computer architecture; as usual in this book, we refer to the 80×86 architecture.

Prehistoric Age: the BIOS

The moment after a computer is powered on, it is practically useless because the RAM chips contain random data and no operating system is running. To begin the boot, a special hardware circuit raises the logical value of the RESET pin of the CPU. After RESET is asserted, some registers of the processor (including cs and eip) are set to fixed values, and the code found at physical address 0xfffffff0 is executed. This address is mapped by the hardware to a certain read-only, persistent memory chip that is often called Read-Only Memory (ROM). The set of programs stored in ROM is traditionally called the *Basic Input/Output System* (*BIOS*) in the 80×86 architecture, because it includes several interrupt-driven low-level procedures used by all operating systems in the booting phase to handle the hardware devices that make up the computer. Some operating systems, such as Microsoft's MS-DOS, rely on BIOS to implement most system calls.

Once in protected mode (see the section "Segmentation in Hardware" in Chapter 2), Linux does not use BIOS any longer, but it provides its own device driver for every

hardware device on the computer. In fact, the BIOS procedures must be executed in real mode, so they cannot share functions even if that would be beneficial.

The BIOS uses Real Mode addresses because they are the only ones available when the computer is turned on. A Real Mode address is composed of a *seg* segment and an *off* offset; the corresponding physical address is given by *seg**16+*off*. As a result, no Global Descriptor Table, Local Descriptor Table, or paging table is needed by the CPU addressing circuit to translate a logical address into a physical one. Clearly, the code that initializes the GDT, LDT, and paging tables must run in Real Mode.

Linux is forced to use BIOS in the bootstrapping phase, when it must retrieve the kernel image from disk or from some other external device. The BIOS bootstrap procedure essentially performs the following four operations:

1. Executes a series of tests on the computer hardware to establish which devices are present and whether they are working properly. This phase is often called *Power-On Self-Test* (*POST*). During this phase, several messages, such as the BIOS version banner, are displayed.

   Recent 80×86, AMD64, and Itanium computers make use of the *Advanced Configuration and Power Interface* (*ACPI*) standard. The bootstrap code in an ACPI-compliant BIOS builds several tables that describe the hardware devices present in the system. These tables have a vendor-independent format and can be read by the operating system kernel to learn how to handle the devices.

2. Initializes the hardware devices. This phase is crucial in modern PCI-based architectures, because it guarantees that all hardware devices operate without conflicts on the IRQ lines and I/O ports. At the end of this phase, a table of installed PCI devices is displayed.

3. Searches for an operating system to boot. Actually, depending on the BIOS setting, the procedure may try to access (in a predefined, customizable order) the first sector (*boot sector*) of every floppy disk, hard disk, and CD-ROM in the system.

4. As soon as a valid device is found, it copies the contents of its first sector into RAM, starting from physical address `0x00007c00`, and then jumps into that address and executes the code just loaded.

The rest of this appendix takes you from the most primitive starting state to the full glory of a running Linux system.

Ancient Age: the Boot Loader

The *boot loader* is the program invoked by the BIOS to load the image of an operating system kernel into RAM. Let's briefly sketch how boot loaders work in IBM's PC architecture.

To boot from a floppy disk, the instructions stored in its first sector are loaded in RAM and executed; these instructions copy all the remaining sectors containing the kernel image into RAM.

Booting from a hard disk is done differently. The first sector of the hard disk, named the *Master Boot Record* (*MBR*), includes the partition table[*] and a small program, which loads the first sector of the partition containing the operating system to be started. Some operating systems, such as Microsoft Windows 98, identify this partition by means of an *active* flag included in the partition table;[†] following this approach, only the operating system whose kernel image is stored in the active partition can be booted. As we will see later, Linux is more flexible because it replaces the rudimentary program included in the MBR with a sophisticated program—the "boot loader"—that allows users to select the operating system to be booted.

Kernel images of earlier Linux versions—up to the 2.4 series—included a minimal "boot loader" program in the first 512 bytes; thus, copying a kernel image starting from the first sector made the floppy bootable. On the other hand, kernel images of Linux 2.6 no longer include such boot loader; thus, in order to boot from floppy disk, a suitable boot loader has to be stored in the first disk sector. Nowadays, booting from a floppy is very similar to booting from a hard disk or from a CD-ROM.

Booting Linux from a Disk

A two-stage boot loader is required to boot a Linux kernel from disk. A well-known Linux boot loader on 80×86 systems is named LInux LOader (LILO). Other boot loaders for 80×86 systems do exist; for instance, the GRand Unified Bootloader (GRUB) is also widely used. GRUB is more advanced than LILO, because it recognizes several disk-based filesystems and is thus capable of reading portions of the boot program from files. Of course, specific boot loader programs exist for all architectures supported by Linux.

LILO may be installed either on the MBR (replacing the small program that loads the boot sector of the active partition) or in the boot sector of every disk partition. In both cases, the final result is the same: when the loader is executed at boot time, the user may choose which operating system to load.

Actually, the LILO boot loader is too large to fit into a single sector, thus it is broken into two parts. The MBR or the partition boot sector includes a small boot loader, which is loaded into RAM starting from address `0x00007c00` by the BIOS. This small program moves itself to the address `0x00096a00`, sets up the Real Mode stack (ranging

---

[*] Each partition table entry typically includes the starting and ending sectors of a partition and the kind of operating system that handles it.

[†] The active flag may be set through programs such as *fdisk*.

from `0x00098000` to `0x000969ff`), loads the second part of the LILO boot loader into RAM starting from address `0x00096c00`, and jumps into it.

In turn, this latter program reads a map of bootable operating systems from disk and offers the user a prompt so she can choose one of them. Finally, after the user has chosen the kernel to be loaded (or let a time-out elapse so that LILO chooses a default), the boot loader may either copy the boot sector of the corresponding partition into RAM and execute it or directly copy the kernel image into RAM.

Assuming that a Linux kernel image must be booted, the LILO boot loader, which relies on BIOS routines, performs essentially the following operations:

1. Invokes a BIOS procedure to display a "Loading" message.

2. Invokes a BIOS procedure to load an initial portion of the kernel image from disk: the first 512 bytes of the kernel image are put in RAM at address `0x00090000`, while the code of the `setup( )` function (see below) is put in RAM starting from address `0x00090200`.

3. Invokes a BIOS procedure to load the rest of the kernel image from disk and puts the image in RAM starting from either low address `0x00010000` (for small kernel images compiled with `make zImage`) or high address `0x00100000` (for big kernel images compiled with `make bzImage`). In the following discussion, we say that the kernel image is "loaded low" or "loaded high" in RAM, respectively. Support for big kernel images uses essentially the same booting scheme as the other one, but it places data in different physical memory addresses to avoid problems with the ISA hole mentioned in the section "Physical Memory Layout" in Chapter 2.

4. Jumps to the `setup( )` code.

Middle Ages: the setup( ) Function

The code of the `setup( )` assembly language function has been placed by the linker at offset `0x200` of the kernel image file. The boot loader can therefore easily locate the code and copy it into RAM, starting from physical address `0x00090200`.

The `setup( )` function must initialize the hardware devices in the computer and set up the environment for the execution of the kernel program. Although the BIOS already initialized most hardware devices, Linux does not rely on it, but reinitializes the devices in its own manner to enhance portability and robustness. `setup( )` performs essentially the following operations:

1. In ACPI-compliant systems, it invokes a BIOS routine that builds a table in RAM describing the layout of the system's physical memory (the table can be seen in the boot kernel messages by looking for the "BIOS-e820" label). In older systems, it invokes a BIOS routine that just returns the amount of RAM available in the system.

2. Sets the keyboard repeat delay and rate. (When the user keeps a key pressed past a certain amount of time, the keyboard device sends the corresponding keycode over and over to the CPU.)

3. Initializes the video adapter card.

4. Reinitializes the disk controller and determines the hard disk parameters.

5. Checks for an IBM Micro Channel bus (MCA).

6. Checks for a PS/2 pointing device (bus mouse).

7. Checks for Advanced Power Management (APM) BIOS support.

8. If the BIOS supports the *Enhanced Disk Drive Services* (*EDD*), it invokes the proper BIOS procedure to build a table in RAM describing the hard disks available in the system. (The information included in the table can be seen by reading the files in the *firmware/edd* directory of the *sysfs* special filesystem.)

9. If the kernel image was loaded low in RAM (at physical address 0x00010000), the function moves it to physical address 0x00001000. Conversely, if the kernel image was loaded high in RAM, the function does not move it. This step is necessary because to be able to store the kernel image on a floppy disk and to reduce the booting time, the kernel image stored on disk is compressed, and the decompression routine needs some free space to use as a temporary buffer following the kernel image in RAM.

10. Sets the A20 pin located on the 8042 keyboard controller. The A20 pin is a hack introduced in the 80286-based systems to make physical addresses compatible with those of the ancient 8088 microprocessors. Unfortunately, the A20 pin must be properly set before switching to Protected Mode, otherwise the 21st bit of every physical address will always be regarded as zero by the CPU. Setting the A20 pin is a messy operation.

11. Sets up a provisional Interrupt Descriptor Table (IDT) and a provisional Global Descriptor Table (GDT).

12. Resets the floating-point unit (FPU), if any.

13. Reprograms the Programmable Interrupt Controllers (PIC) to mask all interrupts, except IRQ2 which is the cascading interrupt between the two PICs.

14. Switches the CPU from Real Mode to Protected Mode by setting the PE bit in the cr0 status register. The PG bit in the cr0 register is cleared, so paging is still disabled.

15. Jumps to the startup_32( ) assembly language function.

Renaissance: the startup_32( ) Functions

There are two different startup_32( ) functions; the one we refer to here is coded in the *arch/i386/boot/compressed/head*.S file. After setup( ) terminates, the function has

been moved either to physical address 0x00100000 or to physical address 0x00001000, depending on whether the kernel image was loaded high or low in RAM.

This function performs the following operations:

1. Initializes the segmentation registers and a provisional stack.

2. Clears all bits in the `eflags` register.

3. Fills the area of uninitialized data of the kernel identified by the `_edata` and `_end` symbols with zeros (see the section "Physical Memory Layout" in Chapter 2).

4. Invokes the `decompress_kernel( )` function to decompress the kernel image. The "Uncompressing Linux…" message is displayed first. After the kernel image is decompressed, the "OK, booting the kernel." message is shown. If the kernel image was loaded low, the decompressed kernel is placed at physical address 0x00100000. Otherwise, if the kernel image was loaded high, the decompressed kernel is placed in a temporary buffer located after the compressed image. The decompressed image is then moved into its final position, which starts at physical address 0x00100000.

5. Jumps to physical address 0x00100000.

The decompressed kernel image begins with another `startup_32( )` function included in the *arch/i386/kernel/head.S* file. Using the same name for both the functions does not create any problems (besides confusing our readers), because both functions are executed by jumping to their initial physical addresses.

The second `startup_32( )` function sets up the execution environment for the first Linux process (process 0). The function performs the following operations:

1. Initializes the segmentation registers with their final values.

2. Fills the bss segment of the kernel (see the section "Program Segments and Process Memory Regions" in Chapter 20) with zeros.

3. Initializes the provisional kernel Page Tables contained in `swapper_pg_dir` and `pg0` to identically map the linear addresses to the same physical addresses, as explained in the section "Kernel Page Tables" in Chapter 2.

4. Stores the address of the Page Global Directory in the `cr3` register, and enables paging by setting the `PG` bit in the `cr0` register.

5. Sets up the Kernel Mode stack for process 0 (see the section "Kernel Threads" in Chapter 3).

6. Once again, the function clears all bits in the `eflags` register.

7. Invokes `setup_idt( )` to fill the IDT with null interrupt handlers (see the section "Preliminary Initialization of the IDT" in Chapter 4).

8. Puts the system parameters obtained from the BIOS and the parameters passed to the operating system into the first page frame (see the section "Physical Memory Layout" in Chapter 2).

9. Identifies the model of the processor.

10. Loads the `gdtr` and `idtr` registers with the addresses of the GDT and IDT tables.

11. Jumps to the `start_kernel( )` function.

Modern Age: the start_kernel( ) Function

The `start_kernel( )` function completes the initialization of the Linux kernel. Nearly every kernel component is initialized by this function; we mention just a few of them:

- The scheduler is initialized by invoking the `sched_init()` function (see Chapter 7).

- The memory zones are initialized by invoking the `build_all_zonelists()` function (see the section "Memory Zones" in Chapter 8).

- The Buddy system allocators are initialized by invoking the `page_alloc_init()` and `mem_init()` functions (see the section "The Buddy System Algorithm" in Chapter 8).

- The final initialization of the IDT is performed by invoking `trap_init()` (see the section "Exception Handling" in Chapter 4) and `init_IRQ()` (see the section "IRQ data structures" in Chapter 4).

- The `TASKLET_SOFTIRQ` and `HI_SOFTIRQ` are initialized by invoking the `softirq_init()` function (see the section "Softirqs" in Chapter 4).

- The system date and time are initialized by the `time_init()` function (see the section "The Linux Timekeeping Architecture" in Chapter 6).

- The slab allocator is initialized by the `kmem_cache_init( )` function (see the section "General and Specific Caches" in Chapter 8).

- The speed of the CPU clock is determined by invoking the `calibrate_delay( )` function (see the section "Delay Functions" in Chapter 6).

- The kernel thread for process 1 is created by invoking the `kernel_thread( )` function. In turn, this kernel thread creates the other kernel threads and executes the */sbin/init* program, as described in the section "Kernel Threads" in Chapter 3.

Besides the "Linux version 2.6.11…" message, which is displayed right after the beginning of `start_kernel( )`, many other messages are displayed in this last phase, both by the *init* program and by the kernel threads. At the end, the familiar login prompt appears on the console (or in the graphical screen, if the X Window System is launched at startup), telling the user that the Linux kernel is up and running.

Modules

As stated in Chapter 1, *modules* are Linux's recipe for effectively achieving many of the theoretical advantages of microkernels without introducing performance penalties.

To Be (a Module) or Not to Be?

When system programmers want to add new functionality to the Linux kernel, they are faced with a basic decision: should they write the new code so that it will be compiled as a module, or should they statically link the new code to the kernel?

As a general rule, system programmers tend to implement new code as a module. Because modules can be linked on demand (as we see later), the kernel does not have to be bloated with hundreds of seldom-used programs. Nearly every higher-level component of the Linux kernel—filesystems, device drivers, executable formats, network layers, and so on—can be compiled as a module. Linux distributions use modules extensively in order to support in a seamless way a wide range of hardware devices. For instance, the distribution puts tens of sound card driver modules in a proper directory, although only one of these modules will be effectively loaded on a specific machine.

However, some Linux code must necessarily be linked statically, which means that either the corresponding component is included in the kernel or it is not compiled at all. This happens typically when the component requires a modification to some data structure or function statically linked in the kernel.

For example, suppose the component has to introduce new fields into the process descriptor. Linking a module cannot change an already defined data structure such as task_struct because, even if the module uses its modified version of the data structure, all statically linked code continues to see the old version. Data corruption easily occurs. A partial solution to the problem consists of "statically" adding the new fields to the process descriptor, thus making them available to the kernel component no matter how it has been linked. However, if the kernel component is never

used, such extra fields replicated in every process descriptor are a waste of memory. If the new kernel component increases the size of the process descriptor a lot, one would get better system performance by adding the required fields in the data structure only if the component is statically linked to the kernel.

As a second example, consider a kernel component that has to replace statically linked code. It's pretty clear that no such component can be compiled as a module, because the kernel cannot change the machine code already in RAM when linking the module. For instance, it is not possible to link a module that changes the way page frames are allocated, because the Buddy system functions are always statically linked to the kernel.[*]

The kernel has two key tasks to perform in managing modules. The first task is making sure the rest of the kernel can reach the module's global symbols, such as the entry point to its main function. A module must also know the addresses of symbols in the kernel and in other modules. Thus, references are resolved once and for all when a module is linked. The second task consists of keeping track of the use of modules, so that no module is unloaded while another module or another part of the kernel is using it. A simple reference count keeps track of each module's usage.

Module Licenses

The license of the Linux kernel (GPL, version 2) is liberal in what users and industries can do with the source code; however, it strictly forbids the release of source code derived from—or heavily depending on—the Linux code under a non-GPL license. Essentially, the kernel developers want to be sure that their code—and all the code derived from it—will remain freely usable by all users.

Modules, however, pose a threat to this model. Someone might release a module for the Linux kernel in binary form only; for instance, a vendor might distribute the driver for its hardware device in a binary-only module. Nowadays, there are quite a few examples of these practices. Theoretically, characteristics and behavior of the Linux kernel might be significantly changed by binary-only modules, thus effectively turning a Linux-derived kernel in a commercial product.

Thus, binary-only modules are not well received by the Linux kernel developer community. The implementation of Linux modules reflect this fact. Basically, each module developer should specify in the module source code the type of license, by using the `MODULE_LICENSE` macro. If the license is not GPL-compatible (or it is not specified at all), the module will not be able to make use of many core functions and data structures of the kernel. Moreover, using a module with a non-GPL license will

---

[*] You might wonder why your favorite kernel component has not been modularized. In most cases, there is no strong technical reason because it is essentially a software license issue. Kernel developers want to make sure that core components will never be replaced by proprietary code released through binary-only "black-box" modules.

"taint" the kernel, which means that any supposed bug in the kernel will not be taken in consideration by the kernel developers.

Module Implementation

Modules are stored in the filesystem as ELF object files and are linked to the kernel by executing the *insmod* program (see the later section, "Linking and Unlinking Modules"). For each module, the kernel allocates a memory area containing the following data:

- A `module` object
- A null-terminated string that represents the name of the module (all modules must have unique names)
- The code that implements the functions of the module

The `module` object describes a module; its fields are shown in Table B-1. A doubly linked circular list collects all `module` objects; the list head is stored in the `modules` variable, while the pointers to the adjacent elements are stored in the `list` field of each `module` object.

Table B-1. The module object

Type	Name	Description
enum module_state	state	The internal state of the module
struct list_head	list	Pointers for the list of modules
char [60]	name	The module name
struct module_kobject	mkobj	Includes a `kobject` data structure and a pointer to this module object
struct module_param_attrs *	param_attrs	Pointer to an array of module parameter descriptors
const struct kernel_symbol *	syms	Pointer to an array of exported symbols
unsigned int	num_syms	Number of exported symbols
const unsigned long *	crcs	Pointer to an array of CRC values for the exported symbols
const struct kernel_symbol *	gpl_syms	Pointer to an array of GPL-exported symbols
unsigned int	num_gpl_syms	Number of GPL-exported symbols
const unsigned long *	gpl_crcs	Pointer to an array of CRC values for the GPL-exported symbols
unsigned int	num_exentries	Number of entries in the module's exception table
const struct exception_table_entry *	extable	Pointer to the module's exception table
int (*)(void)	init	The initialization method of the module

Table B-1. The module object (continued)

Type	Name	Description
void *	module_init	Pointer to the dynamic memory area allocated for module's initialization
void *	module_core	Pointer to the dynamic memory area allocated for module's core functions and data structures
unsigned long	init_size	Size of the dynamic memory area required for module's initialization
unsigned long	core_size	Size of the dynamic memory area required for module's core functions and data structures
unsigned long	init_text_size	Size of the executable code used for module's initialization; used only when linking the module
unsigned long	core_text_size	Size of the core executable code of the module; used only when linking the module
struct mod_arch_specific	arch	Architecture-dependent fields (none in the 80 × 86 architecture)
int	unsafe	Flag set if the module cannot be safely unloaded
int	license_gplok	Flag set if the module license is GPL-compatible
struct module_ref [NR_CPUS]	ref	Per-CPU usage counters
struct list_head	modules_which_use_me	List of modules that rely on this module
struct task_struct *	waiter	The process that is trying to unload the module
void (*)(void)	exit	Exit method of the module
Elf_Sym *	symtab	Pointer to an array of module's ELF symbols for the */proc/kallsyms* file
unsigned long	num_symtab	Number of module's ELF symbols shown in */proc/kallsyms*
char *	strtab	The string table for the module's ELF symbols shown in */proc/kallsyms*
struct module_sect_attrs *	sect_attrs	Pointer to an array of module's section attribute descriptors (displayed in the *sysfs* filesystem)
void *	percpu	Pointer to CPU-specific memory areas
char *	args	Command line arguments used when linking the module

The state field encodes the internal state of the module: it can be MODULE_STATE_LIVE (the module is active), MODULE_STATE_COMING (the module is being initialized), and MODULE_STATE_GOING (the module is being removed).

As already mentioned in the section "Dynamic Address Checking: The Fix-up Code" in Chapter 10, each module has its own exception table. The table includes the addresses of the fixup code of the module, if any. The table is copied into RAM when the module is linked, and its starting address is stored in the extable field of the module object.

Module Usage Counters

Each module has a set of usage counters, one for each CPU, stored in the ref field of the corresponding module object. The counter is increased when an operation involving the module's functions is started and decreased when the operation terminates. A module can be unlinked only if the sum of all usage counters is 0.

For example, suppose that the MS-DOS filesystem layer is compiled as a module and the module is linked at runtime. Initially, the module usage counters are set to 0. If the user mounts an MS-DOS floppy disk, one of the module usage counters is increased by 1. Conversely, when the user unmounts the floppy disk, one of the counters—even different from the one that was increased—is decreased by 1. The total usage counter of the module is the sum of all CPU counters.

Exporting Symbols

When linking a module, all references to global kernel symbols (variables and functions) in the module's object code must be replaced with suitable addresses. This operation, which is very similar to that performed by the linker while compiling a User Mode program (see the section "Libraries" in Chapter 20), is delegated to the *insmod* external program (described later in the section "Linking and Unlinking Modules").

Some special *kernel symbol tables* are used by the kernel to store the symbols that can be accessed by modules together with their corresponding addresses. They are contained in three sections of the kernel code segment: the __kstrtab section includes the names of the symbols, the __ksymtab section includes the addresses of the symbols that can be used by all kind of modules, and the __ksymtab_gpl section includes the addresses of the symbols that can be used by the modules released under a GPL-compatible license. The EXPORT_SYMBOL macro and the EXPORT_SYMBOL_GPL macro, when used inside the statically linked kernel code, force the C compiler to add a specified symbol to the __ksymtab and __ksymtab_gpl sections, respectively.

Only the kernel symbols actually used by some existing module are included in the table. Should a system programmer need, within some module, to access a kernel symbol that is not already exported, he can simply add the corresponding EXPORT_SYMBOL_GPL macro into the Linux source code. Of course, he cannot legally export a new symbol for a module whose license is not GPL-compatible.

Linked modules can also export their own symbols so that other modules can access them. The *module symbol tables* are contained in the __ksymtab, __ksymtab_gpl, and __kstrtab sections of the module code segment. To export a subset of symbols from the module, the programmer can use the EXPORT_SYMBOL and EXPORT_SYMBOL_GPL macros described above. The exported symbols of the module are copied into two memory arrays when the module is linked, and their addresses are stored in the syms and gpl_syms fields of the module object.

Module Dependency

A module (B) can refer to the symbols exported by another module (A); in this case, we say that B is loaded on top of A, or equivalently that A is used by B. To link module B, module A must have already been linked; otherwise, the references to the symbols exported by A cannot be properly linked in B. In short, there is a *dependency* between modules.

The `modules_which_use_me` field of the `module` object of A is the head of a dependency list containing all modules that are used by A; each element of the list is a small `module_use` descriptor containing the pointers to the adjacent elements in the list and a pointer to the corresponding `module` object; in our example, a `module_use` descriptor pointing to the B's `module` object would appear in the `modules_which_use_me` list of A. The `modules_which_use_me` list must be updated dynamically whenever a module is loaded on top of A. The module A cannot be unloaded if its dependency list is not empty.

Beside A and B there could be, of course, another module (C) loaded on top of B, and so on. Stacking modules is an effective way to modularize the kernel source code, thus speeding up its development.

Linking and Unlinking Modules

A user can link a module into the running kernel by executing the *insmod* external program. This program performs the following operations:

1. Reads from the command line the name of the module to be linked.
2. Locates the file containing the module's object code in the system directory tree. The file is usually placed in some subdirectory below */lib/modules*.
3. Reads from disk the file containing the module's object code.
4. Invokes the `init_module( )` system call, passing to it the address of the User Mode buffer containing the module's object code, the length of the object code, and the User Mode memory area containing the parameters of the *insmod* program.
5. Terminates.

The `sys_init_module( )` service routine does all the real work; it performs the following main operations:

1. Checks whether the user is allowed to link the module (the current process must have the `CAP_SYS_MODULE` capability). In every situation where one is adding functionality to a kernel, which has access to all data and processes on the system, security is a paramount concern.

2. Allocates a temporary memory area for the module's object code; then, copies into this memory area the data in the User Mode buffer passed as first parameter of the system call.

3. Checks that the data in the memory area effectively represents a module's ELF object; otherwise, returns an error code.

4. Allocates a memory area for the parameters passed to the *insmod* program, and fills it with the data in the User Mode buffer whose address was passed as third parameter of the system call.

5. Walks the `modules` list to verify that the module is not already linked. The check is done by comparing the names of the modules (`name` field in the `module` objects).

6. Allocates a memory area for the core executable code of the module, and fills it with the contents of the relevant sections of the module.

7. Allocates a memory area for the initialization code of the module, and fills it with the contents of the relevant sections of the module.

8. Determines the address of the `module` object for the new module. An image of this object is included in the `gnu.linkonce.this_module` section of the text segment of the module's ELF file. The `module` object is thus included in the memory area filled in step 6.

9. Stores in the `module_code` and `module_init` fields of the `module` object the addresses of the memory areas allocated in steps 6 and 7.

10. Initializes the `modules_which_use_me` list in the `module` object, and sets to zero all module's reference counters except the counter of the executing CPU, which is set to one.

11. Sets the `license_gplok` flag in the `module` object according to the type of license specified in the module object.

12. Using the kernel symbol tables and the module symbol tables, relocates the module's object code. This means replacing all occurrences of external and global symbols with the corresponding logical address offsets.

13. Initializes the `syms` and `gpl_syms` fields of the `module` object so that they point to the in-memory tables of symbols exported by the module.

14. The exception table of the module (see the section "The Exception Tables" in Chapter 10) is contained in the `__ex_table` section of the module's ELF file, thus it was copied into the memory area allocated in step 6: stores its address in the `extable` field of the `module` object.

15. Parses the arguments of the *insmod* program, and sets the value of the corresponding module variables accordingly.

16. Registers the kobject included in the `mkobj` field of the `module` object so that a new sub-directory for the module appears in the *module* directory of the *sysfs* special filesystem (see the section "Kobjects" in Chapter 13).

17. Frees the temporary memory area allocated in step 2.

18. Adds the `module` object in the `modules` list.

19. Sets the state of the module to `MODULE_STATE_COMING`.

20. If defined, executes the `init` method of the `module` object.

21. Sets the state of the module to `MODULE_STATE_LIVE`.

22. Terminates by returning zero (success).

To unlink a module, a user invokes the *rmmod* external program, which performs the following operations:

1. Reads from the command line the name of the module to be unlinked.

2. Opens the */proc/modules* file, which lists all modules linked into the kernel, and checks that the module to be removed is effectively linked.

3. Invokes the `delete_module()` system call passing to it the name of the module.

4. Terminates.

In turn, the `sys_delete_module()` service routine performs the following main operations:

1. Checks whether the user is allowed to unlink the module (the current process must have the `CAP_SYS_MODULE` capability).

2. Copies the module's name in a kernel buffer.

3. Walks the `modules` list to find the `module` object of the module.

4. Checks the `modules_which_use_me` dependency list of the module; if it is not empty, the function returns an error code.

5. Checks the state of the module; if it is not `MODULE_STATE_LIVE`, returns an error code.

6. If the module has a custom `init` method, the function checks that it has also a custom `exit` method; if no `exit` method is defined, the module should not be unloaded, thus returns an exit code.

7. To avoid race conditions, stops the activities of all CPUs in the system, except the CPU executing the `sys_delete_module()` service routine.

8. Sets the state of the module to `MODULE_STATE_GOING`.

9. If the sum of all reference counters of the module is greater than zero, returns an error code.

10. If defined, executes the `exit` method of the module.

11. Removes the `module` object from the `modules` list, and de-registers the module from the *sysfs* special filesystem.

12. Removes the `module` object from the dependency lists of the modules that it was using.

13. Frees the memory areas that contain the module's executable code, the `module` object, and the various symbol and exception tables.

14. Returns zero (success).

Linking Modules on Demand

A module can be automatically linked when the functionality it provides is requested and automatically removed afterward.

For instance, suppose that the MS-DOS filesystem has not been linked, either statically or dynamically. If a user tries to mount an MS-DOS filesystem, the mount( ) system call normally fails by returning an error code, because MS-DOS is not included in the `file_systems` list of registered filesystems. However, if support for automatic linking of modules has been specified when configuring the kernel, Linux makes an attempt to link the MS-DOS module, and then scans the list of registered filesystems again. If the module is successfully linked, the mount( ) system call can continue its execution as if the MS-DOS filesystem were present from the beginning.

The modprobe Program

To automatically link a module, the kernel creates a kernel thread to execute the *modprobe* external program,[*] which takes care of possible complications due to module dependencies. The dependencies were discussed earlier: a module may require one or more other modules, and these in turn may require still other modules. For instance, the MS-DOS module requires another module named *fat* containing some code common to all filesystems based on a File Allocation Table (FAT). Thus, if it is not already present, the *fat* module must also be automatically linked into the running kernel when the MS-DOS module is requested. Resolving dependencies and finding modules is a type of activity that's best done in User Mode, because it requires locating and accessing module object files in the filesystem.

The *modprobe* external program is similar to *insmod*, since it links in a module specified on the command line. However, *modprobe* also recursively links in all modules used by the module specified on the command line. For instance, if a user invokes *modprobe* to link the MS-DOS module, the program links the *fat* module, if necessary, followed by the MS-DOS module. Actually, *modprobe* simply checks for module dependencies; the actual linking of each module is done by forking a new process and executing *insmod*.

How does *modprobe* know about module dependencies? Another external program named *depmod* is executed at system startup. It looks at all the modules compiled for

---

[*] This is one of the few examples in which the kernel relies on an external program.

the running kernel, which are usually stored inside the */lib/modules* directory. Then it writes all module dependencies to a file named *modules.dep*. The *modprobe* program can thus simply compare the information stored in the file with the list of linked modules yielded by the */proc/modules* file.

The request_module( ) Function

In some cases, the kernel may invoke the `request_module( )` function to attempt automatic linking for a module.

Consider again the case of a user trying to mount an MS-DOS filesystem. If the `get_fs_type( )` function discovers that the filesystem is not registered, it invokes the `request_module( )` function in the hope that MS-DOS has been compiled as a module.

If the `request_module( )` function succeeds in linking the requested module, `get_fs_type( )` can continue as if the module were always present. Of course, this does not always happen; in our example, the MS-DOS module might not have been compiled at all. In this case, `get_fs_type( )` returns an error code.

The `request_module( )` function receives the name of the module to be linked as its parameter. It executes `kernel_thread( )` to create a new kernel thread and waits until that kernel thread terminates.

The kernel thread, in turn, receives the name of the module to be linked as its parameter and invokes the `execve( )` system call to execute the *modprobe* external program,[*] passing the module name to it. In turn, the *modprobe* program actually links the requested module, along with any that it depends on.

---

[*] The name and path of the program executed by `exec_modprobe( )` can be customized by writing into the */proc/sys/kernel/modprobe* file.

Bibliography

This bibliography is broken down by subject area and lists some of the most common and, in our opinion, useful books and online documentation on the topic of kernels.

Books on Unix Kernels

Bach, M. J. *The Design of the Unix Operating System*. Prentice Hall International, Inc., 1986. A classic book describing the SVR2 kernel.

Goodheart, B. and J. Cox. *The Magic Garden Explained: The Internals of the Unix System V Release 4*. Prentice Hall International, Inc., 1994. An excellent book on the SVR4 kernel.

Mauro, J. and R. McDougall. *Solaris Internals: Core Kernel Architecture*. Prentice Hall, 2000. A good introduction to the Solaris kernel.

McKusick, M. K., M. J. Karels, and K. Bostic. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1986. Perhaps the most authoritative book on the 4.4 BSD kernel.

Schimmel, Curt. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley, 1994. An interesting book that deals mostly with the problem of cache implementation in multiprocessor systems.

Vahalia, U. *Unix Internals: The New Frontiers*. Prentice Hall, Inc., 1996. A valuable book that provides plenty of insight on modern Unix kernel design issues. It includes a rich bibliography.

Books on the Linux Kernel

Beck, M., H. Boehme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, and C. Schroter. *Linux Kernel Programming* (3rd ed.). Addison Wesley, 2002. A hardware-independent book covering the Linux 2.4 kernel.

Benvenuti, Christian. *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2006. Covers standard networking protocols and the details of Linux implementation, with a focus on layer 2 and 3 activities.

Corbet, J., A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers (3rd ed.)*. O'Reilly & Associates, Inc., 2005. A valuable book that is somewhat complementary to this one. It gives plenty of information on how to develop drivers for Linux.

Gorman, M. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, 2004. Focuses on a subset of the chapters included in this book, namely those related to the Virtual Memory Manager.

Herbert, T. F. *The Linux TCP/IP Stack: Networking for Embedded Systems (Networking Series)*. Charles River Media, 2004. Describes in great details the TCP/IP Linux implementation in the 2.6 kernel.

Love, R. *Linux Kernel Development (2nd ed.)*. Novell Press, 2005. A hardware-independent book covering the Linux 2.6 kernel. Some readers suggest to read it before attacking this book.

Mosberger, D., S. Eranian, and B. Perens. *IA-64 Linux Kernel: Design and Implementation.* Prentice Hall, Inc., 2002. An excellent description of the hardware-dependent Linux kernel for the Itanium IA-64 microprocessor.

Books on PC Architecture and Technical Manuals on Intel Microprocessors

Intel. *Intel Architecture Software Developer's Manual, vol. 3: System Programming Guide*. 2005. Describes the Intel Pentium microprocessor architecture. It can be downloaded from: *http://developer.intel.com/design/processors/pentium4/manuals/25366816.pdf*.

Intel. *MultiProcessor Specification, Version 1.4*. 1997. Describes the Intel multiprocessor architecture specifications. It can be downloaded from *http://www.intel.com/design/pentium/datashts/24201606.pdf*.

Messmer, H. P. *The Indispensable PC Hardware Book (4th ed.)*. Addison Wesley Professional, 2001. A valuable reference that exhaustively describes the many components of a PC.

Other Online Documentation Sources

Linux source code

The official site for getting kernel source can be found at *http://www.kernel.org*. Many mirror sites are also available all over the world.

A valuable search engine for the Linux 2.6 source code is available at *http://lxr.linux.no*.

GCC manuals

All distributions of the GNU C compiler should include full documentation for all its features, stored in several info files that can be read with the Emacs program or an info reader. By the way, the information on Extended Inline Assembly is quite hard to follow, because it does not refer to any specific architecture. Some pertinent information about 80**x**86 GCC's Inline Assembly and *gas*, the GNU assembler invoked by GCC, can be found at:

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html
http://www.ibm.com/developerworks/linux/library/l-ia.html
http://www.gnu.org/manual/gas-2.9.1/as.html

The Linux Documentation Project

The web site (*http://www.tldp.org*) contains the home page of the Linux Documentation Project, which, in turn, includes several interesting references to guides, FAQs, and HOWTOs.

Linux kernel development forum

The newsgroup *comp.os.linux.development.system* is dedicated to discussions about development of the Linux system.

The linux-kernel mailing list

This fascinating mailing list contains much noise as well as a few pertinent comments about the current development version of Linux and about the rationale for including or not including in the kernel some proposals for changes. It is a living laboratory of new ideas that are taking shape. The name of the mailing list is *linux-kernel@vger.kernel.org*.

The Linux Kernel online book

Authored by David A. Rusling, this 200-page book can be viewed at *http://www.tldp.org/LDP/tlk/tlk.html*, and describes some fundamental aspects of the Linux 2.0 kernel.

Linux Virtual File System

The page at *http://www.safe-mbox.com/~rgooch/linux/docs/vfs.txt* is an introduction to the Linux Virtual File System. The author is Richard Gooch.

Research Papers Related to Linux Development

We list here a few papers that we have mentioned in this book. Needless to say, there are many other articles that have a great impact on the development of Linux.

McCreight, E. "Priority Search Tree," *SIAM J. Comput.*, Vol. 14, No 2, pp. 257-276, May 1985

Johnson, T. and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proceedings of the 20th IEEE VLDB Conf.*, Santiago, Chile, 1994, pp. 439-450.

Bonwick, J. "The Slab Allocator: An Object-Caching Kernel Memory Allocator," *Proceedings of Summer 1994 USENIX Conference*, pp. 87-98.

Source Code Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

W

Index

About the Authors

Daniel P. Bovet received his Ph.D. in computer science at UCLA in 1968 and is a full professor at the University of Rome, Tor Vergata, Italy. He had to wait over 25 years before being able to teach an operating systems course in a proper manner, due to the lack of source code for modern, well-designed systems. Now, thanks to cheap PCs and Linux, Dan and Marco are able to cover all the facets of an operating system and can hand out tough, satisfying homework to their students. (These young students working at home on their PCs are really spoiled; they never had to fight with punched cards.) In fact, Dan was so fascinated by the accomplishments of Linus Torvalds and his followers that he spent the last few years trying to unravel some of Linux's mysteries. It seemed natural, after all that work, to write a book about what he found.

Marco Cesati received a degree in mathematics in 1992 and a Ph.D. in computer science at the University of Rome, La Sapienza, in 1995. He is now a research assistant in the computer science department of the School of Engineering at the University of Rome, Tor Vergata. In the past, he has served as a system administrator and Unix programmer for the university (as a Ph.D. student) and for several institutions (as a consultant). During the last few years, he has been continuously involved in teaching his students how to change the Linux kernel in strange and funny ways.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

Darren Kelly was the production editor for *Understanding the Linux Kernel*, Third Edition. Sharon Lundsford was the copyeditor and Julie Campbell was the proofreader. Mary Brady and Claire Cloutier provided quality control. Jansen Fernald and Loranah Dimant provided production assistance. Amy Parker provided production services.

Edie Freedman designed the cover of this book, based on a series design by herself and Hanna Dyer. The cover image of a man with a bubble is a 19th-century engraving from the Dover Pictorial Archive. Karen Montgomery produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. The chapter opening image is from the Dover Pictorial Archive. This book was converted to FrameMaker 5.5.6 by Keith Fahlgren with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano, Jessamyn Read, and Lesley Borash using Macromedia FreeHand 9 and Adobe Photoshop 6.