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 like task switching and task scheduling.

We continued along this line in the spring semester of 1998, but we moved on to the Linux 2.1 development version. Our course notes were becoming larger and larger. In July, 1998 we contacted O'Reilly & Associates, suggesting they publish a whole book on the Linux kernel. The real work started in the fall of 1998 and lasted about a year and a half. 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 spend some effort deciphering some of the functions that are not, for sake of brevity, fully described.

On another level, the book will give valuable insights 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! Like any good guide, we try to go beyond superficial features. We offer background, such as the history of major features and the reasons they were used.

Organization of the Material

When starting 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 thus requires the study of a few assembly language fragments that interact with the hardware.

When covering hardware features, our strategy will be quite simple: just 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.

The next step consisted of selecting the computer system to be described: 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—thus, on the Intel 80x86 microprocessors and on some support chips included in these personal computers. The term *Intel 80x86 microprocessor* will be used in the forthcoming chapters to denote the Intel 80386, 80486, Pentium, Pentium Pro, Pentium II, and Pentium III microprocessors or compatible models. In a few cases, explicit references will be made to specific models.

One more choice was the order followed in studying Linux components. We tried to follow 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 Intel 80x86 microprocessors in the first part of the book, while the rest of it is relatively hardware-independent. Two significant exceptions are made in Chapter 11, and Chapter 13. In practice, following a bottom-up approach is not as simple as it looks, since the areas of memory management, process management, and filesystem 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 about 4500 C and Assembly files stored in about 270 subdirectories; it consists of about 2 million lines of code, which occupy more than 58 megabytes of disk space. Of course, this book can cover 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 2 megabytes of disk space. Therefore, in order to list all code, without commenting on it, we would need more than 25 books like this![1]

[1] Nevertheless, Linux is a tiny operating system when compared with other commercial giants. Microsoft Windows 2000, for example, reportedly has more than 30 million lines of code. Linux is also small when compared to some popular applications; Netscape Communicator 5 browser, for example, has about 17 million lines of code.

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

- We describe process and memory management fairly thoroughly.
- We cover the Virtual Filesystem and the Ext2 filesystem, 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 a good part of the kernel, as far as the kernel interface is concerned, but do not attempt analysis of any specific driver, including the terminal drivers.
- We do not cover networking, since this area would deserve a whole new book by itself.

In many cases, the original code has been 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).

Overview of the Book

To make life easier, Chapter 1 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 explains how Intel 80x86 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. 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*, which are introduced in Chapter 4. One type of interrupt is crucial for allowing Linux to take care of elapsed time; further details can be found in Chapter 5.

Next we focus again on memory: Chapter 6 describes the sophisticated techniques required to handle the most precious resource in the system (besides the processors, of course), that is, available memory. This resource must be granted both to the Linux kernel and to the user applications. Chapter 7 shows how the kernel copes with the requests for memory issued by greedy application programs.

Chapter 8 explains how a process running in User Mode makes requests to the kernel, while Chapter 9 describes how a process may send synchronization signals to other processes. Chapter 10 explains how Linux executes, in turn, every active process in the system so that all of them can progress toward their completions. Synchronization mechanisms are needed by the kernel too: they are discussed in Chapter 11 for both uniprocessor and multiprocessor systems.

Now we are ready to move on to another essential topic, that is, how Linux implements the filesystem. A series of chapters covers this topic: Chapter 12 introduces a general layer that supports many different filesystems. Some Linux files are special because they provide trapdoors to reach hardware devices; Chapter 13 offers insights on these special files and on the corresponding hardware device drivers. Another issue to be considered is disk access time; Chapter 14 shows how a clever use of RAM reduces disk accesses and thus improves system performance significantly. Building on the material covered in these last chapters, we can now explain in Chapter 15, how user applications access normal files. Chapter 16 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 17, which illustrates the most-used Linux filesystem, namely Ext2.

The last two chapters end our detailed tour of the Linux kernel: Chapter 18 introduces communication mechanisms other than signals available to User Mode processes; Chapter 19 explains how user applications are started.

Last but not least are the appendixes: Appendix A sketches out how Linux is booted, while Appendix B describes how to dynamically reconfigure the running kernel, adding and removing functionalities as needed. Appendix C is just a list of the directories that contain the Linux source code. The Source Code Index includes all the Linux symbols referenced in the book; you will find here 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 Assembly language.

Conventions in This Book

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

`Constant Width`

Is used to show the contents of code files or the output from commands, and to indicate source code keywords that appear in code.

Italic

Is used for file and directory names, program and command names, command-line options, URLs, and for emphasizing new terms.

How to Contact Us

We have tested and verified all the information in this book to the best of our abilities, but you may find that features have changed or that we have let errors slip through the production of the book. Please let us know of any errors that you find, as well as suggestions for future editions, by writing to:

O'Reilly & Associates, Inc. 101 Morris St. Sebastopol, CA 95472 (800) 998-9938 (in the U.S. or Canada) (707) 829-0515 (international/local) (707) 829-0104 (fax)

You can also send messages electronically. To be put on our mailing list or to request a catalog, send email to:

info@oreilly.com

To ask technical questions or to comment on the book, send email to:

bookquestions@oreilly.com

We have a web site for the book, where we'll list reader reviews, errata, and any plans for future editions. You can access this page at:

http://www.oreilly.com/catalog/linuxkernel/

We also have an additional web site where you will find material written by the authors about the new features of Linux 2.4. Hopefully, this material will be used for a future edition of this book. You can access this page at:

http://www.oreilly.com/catalog/linuxkernel/updates/

For more information about this book and others, see the O'Reilly web site:

http://www.oreilly.com/

Acknowledgments

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

Andy Oram, our wonderful editor at O'Reilly & Associates, 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.

Many thanks also to the O'Reilly staff, especially Rob Romano, the technical illustrator, and Lenny Muellner, for tools support.

We had some prestigious reviewers who read our text quite carefully (in alphabetical order by first name): Alan Cox, Michael Kerrisk, Paul Kinzelman, Raph Levien, and Rik van Riel. Their comments helped us to remove several errors and inaccuracies and have made this book stronger.

—Daniel P. Bovet, Marco Cesati

September 2000

Chapter 1. Introduction

Linux is a member of the large family of Unix-like operating systems. A relative newcomer 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, which is now owned by Novell; the 4.4 BSD release from the University of California at Berkeley (4.4BSD), Digital Unix from Digital Equipment Corporation (now Compaq); AIX from IBM; HP-UX from Hewlett-Packard; and Solaris from Sun Microsystems.

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 Alpha, SPARC, Motorola MC680x0, PowerPC, and IBM System/390.

One of the more appealing benefits to Linux is that it isn't a commercial operating system: its source code under the GNU Public License[1] 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.

[1] 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 applications such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on. However, since most of these programs are freely available under the GNU General Public License, they can be installed into one of the filesystems supported by Linux.

Since Linux is a kernel, 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 FTP sites. The Linux source code is usually installed in the */usr/src/linux* directory. In the rest of this book, all file pathnames will refer implicitly to that directory.

1.1 Linux Versus Other Unix-Like Kernels

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

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.[2]

[2] As a matter of fact, several non-Unix operating systems like Windows NT are POSIX-compliant.

In order 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. What you read in this book and see in the Linux kernel, therefore, may help you understand the other Unix variants too.

The 2.2 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, like virtual memory, a virtual filesystem, lightweight processes, reliable signals, SVR4 interprocess communications, support for Symmetric Multiprocessor (SMP) systems, and so on.

By itself, the Linux kernel is not very innovative. 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 described in Bach's book (i.e., SVR4). However, Linux doesn't stick to any particular variant. Instead, it tries to adopt good features and design choices of several different Unix kernels.

Here is an assessment of how Linux competes against some well-known commercial Unix kernels:

- The Linux kernel is monolithic**.** 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. A notable exception is Carnegie-Mellon's Mach 3.0, which follows a microkernel approach.
- Traditional Unix kernels are compiled and linked statically**.** 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, since it is able to automatically load and unload modules on demand. Among the main commercial Unix variants, only the SVR4.2 kernel has a similar feature.
- Kernel threading**.** Some modern Unix kernels, like Solaris 2.x 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, since 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; since Linux kernel threads cannot execute user programs, 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 well designed in terms of many relatively independent execution flows sharing a large portion of the application data structures. A multithreaded user application could be composed of many *lightweight processes* (LWP), or 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.

- Linux is a nonpreemptive kernel. This means that Linux cannot arbitrarily interleave execution flows while they are in privileged mode. Several sections of kernel code assume they can run and modify data structures without fear of being interrupted and having another thread alter those data structures. Usually, fully preemptive kernels are associated with special real-time operating systems. Currently, among conventional, general-purpose Unix systems, only Solaris 2.x 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 2.2 offers an evolving kind of support for symmetric multiprocessing (SMP), which means not only that the system can use multiple processors but also that any processor can handle any task; there is no discrimination among them. However, Linux 2.2 does not make optimal use of SMP. Several kernel activities that could be executed concurrently—like filesystem handling and networking—must now be executed sequentially.
- Filesystem. Linux's standard filesystem lacks some advanced features, such as journaling. However, more advanced filesystems for Linux are available, although not included in the Linux source code; among them, 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 a relatively easy task.
- STREAMS**.** Linux has no analog to the STREAMS I/O subsystem introduced in SVR4, although it is included nowadays in most Unix kernels and it has become the preferred interface for writing device drivers, terminal drivers, and network protocols.

This somewhat disappointing assessment does not depict, however, the whole truth. Several features make Linux a wonderfully unique operating system. Commercial Unix kernels often introduce new features in order 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 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 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 General Public License (GPL), you are allowed to freely read and modify the source code of the kernel and of all system programs.[3]

[3] Several commercial companies have started to support their products under Linux, most of which aren't distributed under a GNU Public License. Therefore, you may not be allowed to read or modify their source code.

Linux runs on low-end, cheap hardware platforms.

You can even build a network server using an old Intel 80386 system with 4 MB of RAM.

Linux is powerful.

Linux systems are very fast, since they fully exploit the features of the hardware components. The main Linux target 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 has a high standard for source code quality.

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

The Linux kernel can be very small and compact.

Indeed, it is possible to fit both a kernel image and full root filesystem, including all fundamental system programs, on just one 1.4 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.

It lets you directly mount filesystems for all versions of MS-DOS and MS Windows, SVR4, OS/2, Mac OS, Solaris, SunOS, NeXTSTEP, many BSD variants, and so on. Linux is also able to operate with many network layers like Ethernet, Fiber Distributed Data Interface (FDDI), High Performance Parallel Interface (HIPPI), IBM's Token Ring, AT&T WaveLAN, DEC RoamAbout DS, and so forth. 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 applications written for MS-DOS, MS Windows, SVR3 and R4, 4.4BSD, SCO Unix, XENIX, and others on the Intel 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 the Microsoft ones.

> Therefore, all commercial Unix variants run on a restricted subset of hardware components.

With an estimated installed base of more than 12 million and growing, people who are used to certain creature features that are standard under other operating systems are starting to expect the same from Linux. As such, the demand on Linux developers is also increasing. Luckily, though, Linux has evolved under the close direction of Linus over the years, to accommodate the needs of the masses.

1.2 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 nine subdirectories corresponding to the nine hardware platforms supported. The standard names of the platforms are:

arm

> Acorn personal computers

alpha

> Compaq Alpha workstations

i386

> IBM-compatible personal computers based on Intel 80x86 or Intel 80x86-compatible microprocessors

m68k

> Personal computers based on Motorola MC680x0 microprocessors

mips

> Workstations based on Silicon Graphics MIPS microprocessors

ppc

> Workstations based on Motorola-IBM PowerPC microprocessors

sparc

> Workstations based on Sun Microsystems SPARC microprocessors

sparc64

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

s390

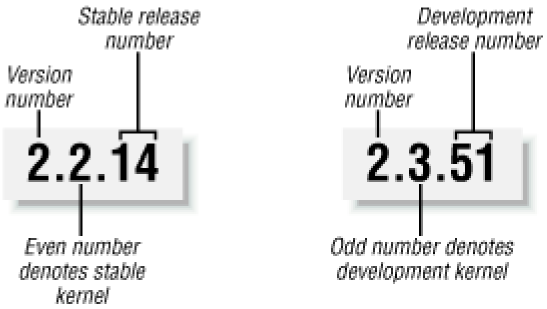
IBM System/390 mainframes

1.3 Linux Versions

Linux distinguishes stable kernels from development kernels through a simple numbering scheme. Each version is characterized by three numbers, separated by periods. The first two numbers are used to identify the version; the third number identifies the release.

As shown in Figure 1-1, if the second number is even, it denotes a stable kernel; otherwise, it denotes a development kernel. At the time of this writing, the current stable version of the Linux kernel is 2.2.14, and the current development version is 2.3.51. The 2.2 kernel, which is the basis for this book, was first released in January 1999, and it differs considerably from the 2.0 kernel, particularly with respect to memory management. Work on the 2.3 development version started in May 1999.

Figure 1-1. Numbering Linux versions


Stable release number
Version number
2.2.14
Even number denotes stable kernel
Development release number
Version number
2.3.51
Odd number denotes development kernel

New releases of a stable version come out mostly to fix bugs reported by users. The main algorithms and data structures used to implement the kernel are left unchanged.

Development versions, on the other hand, may differ quite significantly from one another; kernel developers are free to experiment with different solutions that occasionally lead to drastic kernel changes. Users who rely on development versions for running applications may experience unpleasant surprises when upgrading their kernel to a newer release. This book concentrates on the most recent stable kernel that we had available because, among all the new features being tried in experimental kernels, there's no way of telling which will ultimately be accepted and what they'll look like in their final form.

At the time of this writing, Linux 2.4 has not officially come out. We tried to anticipate the forthcoming features and the main kernel changes with respect to the 2.2 version by looking at the Linux 2.3.99-pre8 prerelease. Linux 2.4 inherits a good deal from Linux 2.2: many concepts, design choices, algorithms, and data structures remain the same. For that reason, we conclude each chapter by sketching how Linux 2.4 differs from Linux 2.2 with respect to the topics just discussed. As you'll notice, the new Linux is gleaming and shining; it should appear more appealing to large corporations and, more generally, to the whole business community.

1.4 Basic Operating System Concepts

Any 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, then, is where we fix our attention in this book. Hence, we'll 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 make use of 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 relative hardware components on behalf of the user program.

In order 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 Unix-like systems, thus also to Linux. The other chapters of this book will hopefully help you to understand the Linux kernel internals.

1.4.1 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 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

In order to ensure safe protection mechanisms, operating systems must make use of 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.

1.4.2 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 operating system asks for a *login name* and a *password*. If the user does not input a valid pair, the system denies access. Since the password is assumed to be secret, the user's privacy is ensured.

In order to selectively share material with other users, each user is a member of one or more *groups*, which are identified by a unique number called a *Group ID* , or GID. Each file is also associated with exactly one group. For example, access could be set so that 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*, *superuser*, or *supervisor*. The system administrator must log in as root in order to handle user accounts, perform maintenance tasks like system backups and program upgrades, and so on. The root user can do almost everything, since 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 interfere with the activity of every running user program.

1.4.3 Processes

All operating systems make use of 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*.[4] 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.

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

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 the same time. The choice of the process that can progress is left to an operating system component called the *scheduler*. Some operating systems allow only *nonpreemptive* processes, which means that the scheduler is invoked only when a process voluntarily relinquishes the CPU. But processes of a multiuser system must be *preemptive* ; the operating system tracks how long each process holds the CPU and periodically activates the scheduler.

Unix is a multiprocessing operating system with preemptive processes. Indeed, the process abstraction is really fundamental in all Unix systems. 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 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), 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.

1.4.4 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, system call handlers, and so on.

Although academic research on operating systems is oriented toward microkernels, such operating systems are generally slower than monolithic ones, since 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, since any 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 fairly easily ported to other architectures, since 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, since system processes that aren't implementing needed functionalities might be swapped out or destroyed.

Modules are a kernel feature that effectively achieves many of the theoretical advantages of microkernels without introducing performance penalties. 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:

Modularized approach

Since 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 Compaq'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 mechanism also can be made transparent to the user, since linking and unlinking can be performed automatically by the kernel.

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.[5]

[5] A small performance penalty occurs when the module is linked and when it is unlinked. However, this penalty can be compared to the penalty caused by the creation and deletion of system processes in microkernel operating systems.

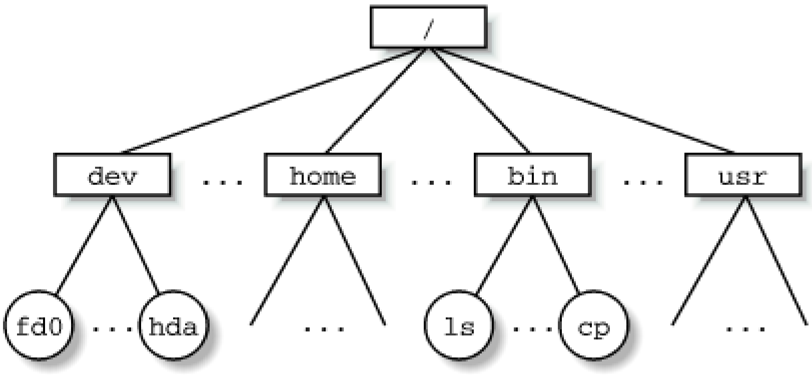
1.5 An Overview of the Unix Filesystem

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.

1.5.1 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 name space as shown in Figure 1-2.

Figure 1-2. 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,[6] 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.

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

Unix associates a *current working directory* with each process (see Section 1.6.1 later in this chapter); it belongs to the process execution context, and it identifies the directory currently used by the process. In order 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 a*bsolute*, since 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*, since 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.

1.5.2 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, thus several filenames.

The Unix command:

```
$ ln f1 f2
```

is used to create a new hard link that has the pathname `f2` for a file identified by the pathname `f1`.

Hard links have two limitations:

- Users are not allowed to create hard links for directories. This 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 since 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.

In order to overcome these limitations, *soft links* (also called *symbolic links*) have been introduced. Symbolic links are short files that contain an arbitrary pathname of another file. The pathname may refer to any file located in any filesystem; it may even refer to a nonexistent file.

The Unix command:

```
$ ln -s f1 f2
```

creates a new soft link with pathname `f2` that refers to pathname `f1`. When this command is executed, the filesystem creates a soft link and writes into it the `f1` pathname. It then inserts—in the proper directory—a new entry containing the last name of the `f2` pathname. In this way, any reference to `f2` can be translated automatically into a reference to `f1`.

1.5.3 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 will be described in detail in Chapter 17.

Device files are related to I/O devices and 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 Section 1.6.5 later in this chapter and Chapter 18).

1.5.4 File Descriptor and Inode

Unix makes a clear distinction between a file and a file descriptor. With the exception of device and special files, each file consists of a sequence of characters. 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 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
- User ID of the file owner
- 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 next section)

1.5.5 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 Group ID (GID) of the process group. However, if the executable file has the `sgid` flag set, the process gets the ID of the file group.

`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.[7]

[7] This flag has become obsolete; other approaches based on sharing of code pages are now used (see Chapter 7).

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

1.5.6 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. Since 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. Whenever a process wants to perform some operation on a specific file, it uses the proper system call and passes the file pathname as a parameter.

All Unix kernels devote great attention to the efficient handling of hardware block devices in order 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. In order to understand those descriptions, you will need to know how the main file-handling system calls are used; they are described in the next section.

1.5.6.1 Opening a file

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

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

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 can also 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 *file descriptor* . An open file object contains:

- Some file-handling data structures, like a pointer to the kernel buffer memory area where file data will be copied; 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 is enabled to invoke. The set of permitted functions depends on the value of the `flag` parameter.

We'll 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.
- 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).

In order to create a new file, the process may also invoke the `create( )` system call, which is handled by the kernel exactly like `open( )`.

1.5.6.2 Accessing an opened file

Regular Unix files can be addressed either sequentially or randomly, while device files and named pipes are usually accessed sequentially (see Chapter 13). 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. In order 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 (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 meaning:

`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 be 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 is also updated by adding `nread` to its previous value. The `write( )` parameters are similar.

1.5.6.3 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 still opened files.

1.5.6.4 Renaming and deleting a file

In order 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);
```

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

1.6 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.

1.6.1 The Process/Kernel Model

As already mentioned, a CPU can run either in User Mode or in Kernel Mode. Actually, some CPUs can have more than two execution states. For instance, the Intel 80x86 microprocessors have four different execution states. But all standard Unix kernels make use of 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 executes most of the time 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.

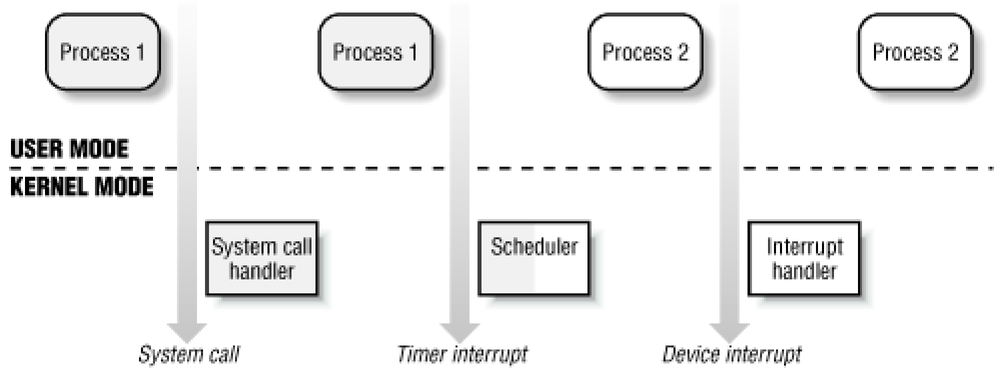
The kernel itself is not a process but a process manager. The process/kernel model assumes that processes that require a kernel service make use of specific programming constructs called *system calls*. Each system call sets up the group of parameters that identifies the process request and then executes 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.

Notice how the process/ kernel model is somewhat orthogonal to the CPU state: on a uniprocessor system, only one process is running at any 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 1-3 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 1-3. 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 some 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*. Since peripheral devices operate asynchronously with respect to the CPU, interrupts occur at unpredictable times.
- A kernel thread is executed; since it runs in Kernel Mode, the corresponding program must be considered part of the kernel, albeit encapsulated in a process.

1.6.2 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. Since the stored value of the program counter points to the instruction following the last instruction executed, the process resumes execution from 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.

1.6.3 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 of them can be blocked in Kernel Mode waiting for the CPU or the completion of some I/O operation. For instance, after issuing a read to a disk on behalf of some process, the kernel will let the disk controller handle it and will resume 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 just 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. Every process in Kernel Mode acts on its own set of memory locations and cannot interfere with the others.

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, since 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.

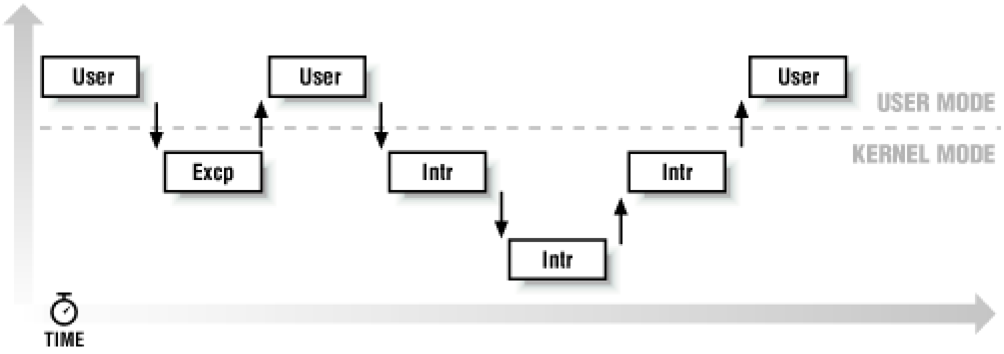
In the simplest case, the CPU executes a kernel control path sequentially from the first instruction to the last. When one of the following events occurs, however, the CPU interleaves the kernel control paths:

- 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, an 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 could 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 elapsed system time is accounted to it. However, the interrupt handler doesn't necessarily operate on behalf of the process.

Figure 1-4 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 1-4. Interleaving of kernel control paths



1.6.4 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 area and makes use of another stack.

Since 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 will be 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 can also 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 memory residing on a 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.

1.6.5 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 will 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, let's suppose that a global variable V contains the number of available items of some system resource. A 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 decrements 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 decrement V and take the resource item, which B already uses. As a final result, V contains -1, and two kernel control paths are using the same resource item with potentially disastrous effects.

When the outcome of some 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 would not be possible if the two control paths read and decrement 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*.[8]

[8] Synchronization problems have been fully described in other works; we refer the interested reader to books on the Unix operating systems (see the bibliography near the end of the book).

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 will concentrate on how to synchronize kernel control paths.

1.6.5.1 Nonpreemptive kernels

In search of a drastically simple solution to synchronization problems, most 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.

Nonpreemptability is ineffective in multiprocessor systems, since two kernel control paths running on different CPUs could concurrently access the same data structure.

1.6.5.2 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 this mechanism doesn't work at all. There is no way to ensure that no other CPU can access the same data structures updated in the protected critical region.

1.6.5.3 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; the semaphore 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 decrements 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 increments 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.

1.6.5.4 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. Since 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 make use of *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 is that the system hangs.

1.6.5.5 Avoiding deadlocks

Processes or kernel control paths that synchronize with other control paths may easily enter in a *deadlocked* 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 waitings among groups of processes may also 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, deadlock becomes an issue when the number of kernel semaphore types 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 introducing a very limited number of semaphore types and by requesting semaphores in an ascending order.

1.6.6 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 `SIGTERM` to a foreground process by pressing the interrupt keycode (usually, CTRL-C) at the terminal.

Synchronous errors or exceptions

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

The POSIX standard defines about 20 different signals, two 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 reception 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 since the POSIX semantics allows processes to temporarily block signals. Moreover, a few signals such as `SIGKILL` cannot be directly handled by the process and cannot be 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 Section 1.6.5 earlier in this chapter, except that they are reserved for processes in User Mode. Message queues allow processes to exchange messages by making use of the `msgsnd( )` and `msgget( )` system calls, which respectively insert a message into a specific message queue and extract a message from it.

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.

1.6.7 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 has been 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 each other 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 assign the copies 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 Section 7.4.4 in Chapter 7.

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.

1.6.7.1 Zombie processes

How can a parent process inquire about termination of its children? The `wait( )` 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 `wait( )` system call on it. The system call handler extracts some data about resource usage from the process descriptor fields; the process descriptor may be released once the data has been collected. If no child process has already terminated when the `wait( )` 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 `wait( )` system calls are also quite common.

It's a good practice for the kernel to keep around information on a child process until the parent issues its `wait( )` 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* that 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 `wait( )` system calls, whose side effect is to get rid of all zombies.

1.6.7.2 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 *process group ID* field. 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.

1.6.8 Memory Management

Memory management is by far the most complex activity in a Unix kernel. We shall dedicate more than a third of this book just to describing how Linux does it. This section illustrates some of the main issues related to memory management.

1.6.8.1 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, since 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,[9] the kernel and the MMU cooperate to locate the actual physical location of the requested memory item.

[9] These addresses have different nomenclatures depending on the computer architecture. As we'll see in Chapter 2, Intel 80x86 manuals refer to them as "logical addresses."

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* 4 or 8 KB in length, and a set of page tables is introduced to specify the correspondence between virtual and physical addresses. These circuits make memory allocation simpler, since a request for a block of contiguous virtual addresses can be satisfied by allocating a group of page frames having noncontiguous physical addresses.

1.6.8.2 Random access memory usage

All Unix operating systems clearly distinguish 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
- To get better performance from disks and other buffered devices by means of caches

Each request type is valuable. On the other hand, since 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 shall see in Chapter 16, 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.

1.6.8.3 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 will come from other kernel subsystems needing memory for kernel use, and some requests will 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, since 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 in order to borrow and release page frames from them.

Several kinds of KMAs have been proposed, which are based on a variety of different algorithmic techniques, including:

- 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 shall see in Chapter 6, Linux's KMA uses a Slab allocator on top of a Buddy system.

1.6.8.4 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 (that is, 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 some 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 it marks them read only. An exception is raised as soon 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.

1.6.8.5 Swapping and caching

In order to extend the size of the virtual address space usable by the processes, the Unix operating system makes use of *swap areas* on disk. The virtual memory system regards the contents of a page frame as the basic unit for swapping. Whenever some process refers to a swapped-out page, the MMU raises an exception. The exception handler then allocates a new page frame and initializes the page frame with its old contents saved on disk.

On the other hand, physical memory is also 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 by loading into RAM a set of disk buffers corresponding to blocks read from 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. In order to avoid data loss, all operating systems take care to periodically write dirty buffers back to disk.

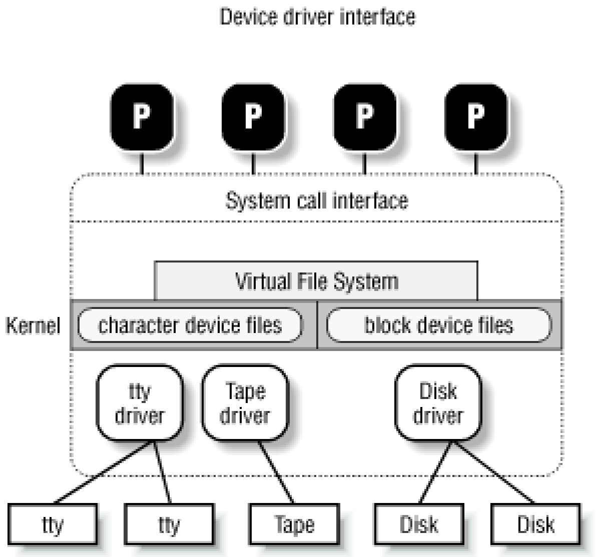
1.6.9 Device Drivers

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 a 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, thus minimizing the size of the kernel image stored in RAM.

Figure 1-5 illustrates how device drivers interface with the rest of the kernel and with the processes. 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 in order to perform the requested operation on the hardware component.

Figure 1-5. Device driver interface



It is worth mentioning that at the time Unix was introduced graphical terminals were uncommon and expensive, and thus 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 2.2, include limited support for some frame buffer devices, thus allowing a program to access the local memory inside a video card through a device file.

Chapter 2. 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 in case of programming errors.

As in the rest of this book, we offer details in this chapter on how Intel 80x86 microprocessors address memory chips and how Linux makes use of 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 6, discusses how the kernel allocates main memory to itself, while Chapter 7, considers how linear addresses are assigned to processes.

2.1 Memory Addresses

Programmers casually refer to a *memory address* as the way to access the contents of a memory cell. But when dealing with Intel 80x86 microprocessors, we have to distinguish among 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 Intel 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

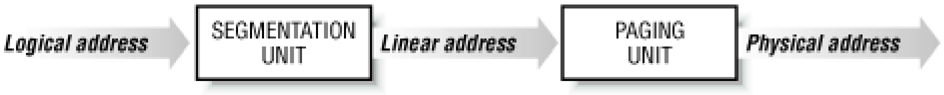
A single 32-bit unsigned integer that can be used to address up to 4 GB, that is, up to 4,294,967,296 memory cells. Linear addresses are usually represented in hexadecimal notation; their values range from `0x00000000` to `0xffffffff`.

Physical address

Used to address memory cells included 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 unsigned integers.

The CPU control unit transforms a logical address into a linear address by means of a hardware circuit called a *segmentation unit*; successively, 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



2.2 Segmentation in Hardware

Starting with the 80386 model, Intel microprocessors perform address translation in two different ways called *real mode* and *protected mode*. 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). We shall thus focus our attention on protected mode.

2.2.1 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 *Segment Selector*, while the offset is a 32-bit field.

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 static and external data

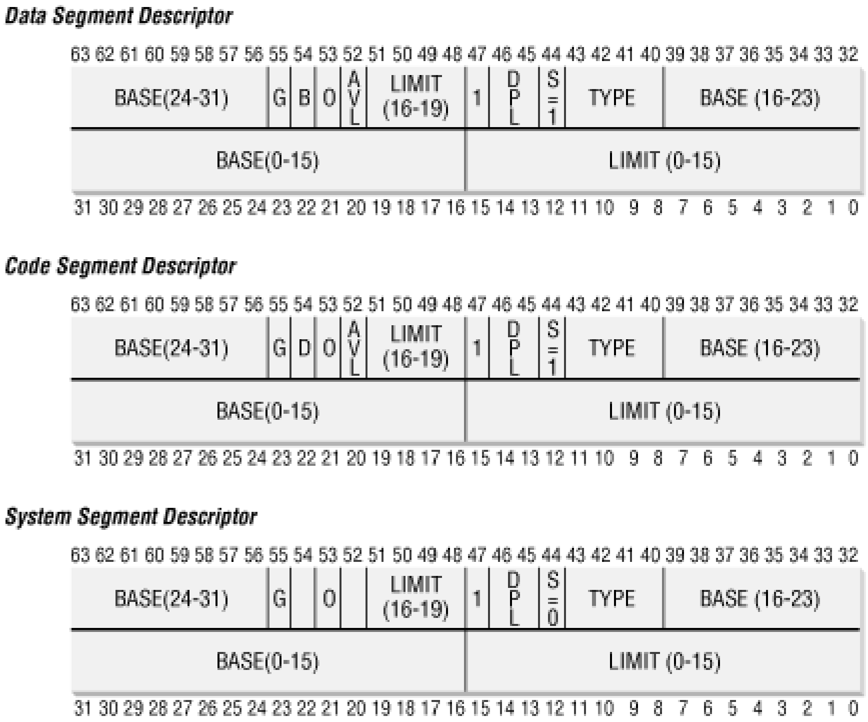
The remaining three segmentation registers are general purpose and may refer to arbitrary segments.

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 denotes the highest privilege level, while the value 3 denotes the lowest one. Linux uses only levels and 3, which are respectively called Kernel Mode and User Mode.

2.2.2 Segment Descriptors

Each segment is represented by an 8-byte *Segment Descriptor* (see Figure 2-2) that describes the segment characteristics. Segment Descriptors are stored either in the *Global Descriptor Table* (*GDT*) or in the *Local Descriptor Table* (*LDT*).

Figure 2-2. Segment Descriptor format



Usually only one GDT is defined, while each process may have its own LDT. The address of the GDT in main memory is contained in the `gdtr` processor register and the address of the currently used LDT is contained in the `ldtr` processor register.

Each Segment Descriptor consists of the following fields:

- A 32-bit `Base` field that contains the linear address of the first byte of the segment.
- A `G` granularity flag: if it is cleared, the segment size is expressed in bytes; otherwise, it is expressed in multiples of 4096 bytes.
- A 20-bit `Limit` field that denotes the segment length in bytes. If `G` is set to 0, the size of a non-null segment may vary between 1 byte and 1 MB; otherwise, it may vary between 4 KB and 4 GB.
- An `S` system flag: if it is cleared, the segment is a system segment that stores kernel data structures; otherwise, it is a normal code or data segment.
- A 4-bit `Type` field that characterizes the segment type and its access rights. The following Segment Descriptor types are widely used:

  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.

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 Section 3.2.2 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 the CPU. The `S` flag of such descriptors is set to 0.

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.

- A `DPL` (*Descriptor Privilege Level*) 2-bit field 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 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.
- A `Segment-Present` flag that is set to if the segment is currently not stored in main memory. Linux always sets this field to 1, since it never swaps out whole segments to disk.
- An additional flag 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 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).
- A reserved bit (bit 53) always set to 0.
- An `AVL` flag that may be used by the operating system but is ignored in Linux.

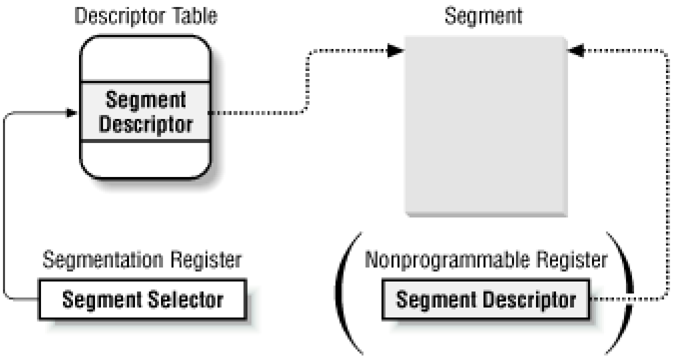
2.2.3 Segment Selectors

To speed up the translation of logical addresses into linear addresses, the Intel 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 just refer directly to the CPU register containing the Segment Descriptor. Accesses to the GDT or LDT are necessary only when the contents of the segmentation register change (see Figure 2-3). Each Segment Selector includes the following fields:

- A 13-bit index (described further in the text following this list) that identifies the corresponding Segment Descriptor entry contained in the GDT or in the LDT
- A `TI` (*Table Indicator*) flag that specifies whether the Segment Descriptor is included in the GDT (`TI` = 0) or in the LDT (`TI` = 1)
- An `RPL` (*Requestor Privilege Level*) 2-bit field, which is precisely the Current Privilege Level of the CPU when the corresponding Segment Selector is loaded into the `cs` register[1]

[1] The RPL field may also be used to selectively weaken the processor privilege level when accessing data segments; see Intel documentation for details.

Figure 2-3. Segment Selector and Segment Descriptor



Since a Segment Descriptor is 8 bytes long, its relative address inside the GDT or the LDT is obtained by multiplying the most significant 13 bits 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 x 8)`, or `0x00020010`.

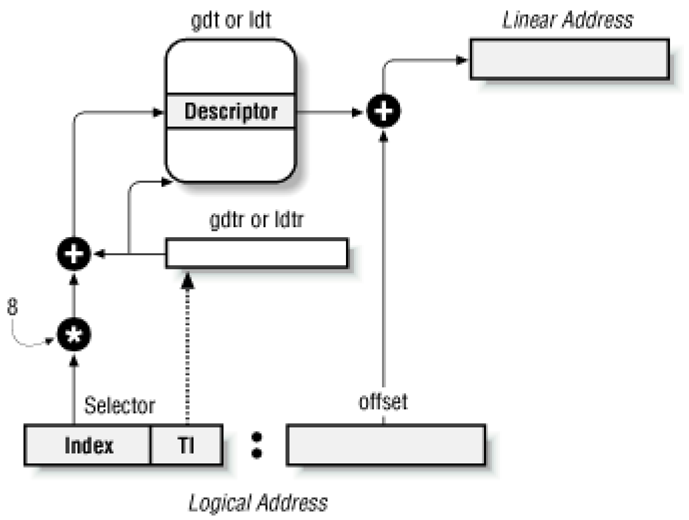
The first entry of the GDT is always set to 0: this ensures that logical addresses with a null Segment Selector will be considered invalid, thus causing a processor exception. The maximum number of Segment Descriptors that can be stored in the GDT is thus 8191, that is, 2^{13}-1.

2.2.4 Segmentation Unit

Figure 2-4 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, in order 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 to the `Base` field of the Segment Descriptor the offset of the logical address, thus obtains the linear address.

Figure 2-4. 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.

2.3 Segmentation in Linux

Segmentation has been included in Intel microprocessors to encourage programmers to split their applications in 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 since both can be used to separate the physical address spaces of processes: segmentation can assign a different 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 register values, that is, when they share the same set of linear addresses.
- One of the design objectives of Linux is portability to the most popular architectures; however, several RISC processors support segmentation in a very limited way.

The 2.2 version of Linux uses segmentation only when required by the Intel 80x86 architecture. In particular, all processes use the same logical addresses, so the total number of segments to be defined is quite limited and it is possible to store all Segment Descriptors in the Global Descriptor Table (GDT). This table is implemented by the array `gdt_table` referred by the `gdt` variable. 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 appendix so you can quickly find it in the source code.

Local Descriptor Tables are not used by the kernel, although a system call exists that allows processes to create their own LDTs. This turns out to be useful to applications such as Wine that execute segment-oriented Microsoft Windows applications.

Here are the segments used by Linux:

- A kernel code segment. The fields of the corresponding Segment Descriptor in the GDT have the following values:
    - `Base` = `0x00000000`
    - `Limit` = `0xfffff`
    - `G` (granularity flag) = `1`, for segment size expressed in pages
    - `S` (system flag) = `1`, for normal code or data segment
    - `Type` = `0xa`, for code segment that can be read and executed
    - `DPL` (Descriptor Privilege Level) = `0`, for Kernel Mode
    - `D/B` (32-bit address flag) = `1`, for 32-bit offset addresses

  Thus, the linear addresses associated with that segment start at and reach the addressing limit of 2^{32} - 1. The `S` and `Type` fields specify that the segment is a code segment that can be read and executed. Its `DPL` value is 0, thus it can be accessed only in Kernel Mode. The corresponding Segment Selector is defined by the `__KERNEL_CS` macro: in order to address the segment, the kernel just loads the value yielded by the macro into the `cs` register.

- A kernel data segment. The fields of the corresponding Segment Descriptor in the GDT have the following values:
    - `Base` = `0x00000000`
    - `Limit` = `0xfffff`
    - `G` (granularity flag) = `1`, for segment size expressed in pages
    - `S` (system flag) = `1`, for normal code or data segment
    - `Type` = `2`, for data segment that can be read and written
    - `DPL` (Descriptor Privilege Level) = `0`, for Kernel Mode
    - `D/B` (32-bit address flag) = `1`, for 32-bit offset addresses

  This segment is identical to the previous one (in fact, they overlap in the linear address space) except for the value of the `Type` field, which specifies that it is a data segment that can be read and written. The corresponding Segment Selector is defined by the `__KERNEL_DS` macro.

- A user code segment shared by all processes in User Mode. The fields of the corresponding Segment Descriptor in the GDT have the following values:
    - `Base` = `0x00000000`
    - `Limit` = `0xfffff`
    - `G` (granularity flag) = `1`, for segment size expressed in pages
    - `S` (system flag) = `1`, for normal code or data segment
    - `Type` = `0xa`, for code segment that can be read and executed
    - `DPL` (Descriptor Privilege Level) = `3`, for User Mode
    - `D/B` (32-bit address flag) = `1`, for 32-bit offset addresses

  The `S` and `DPL` fields specify that the segment is not a system segment and that its privilege level is equal to 3; it can thus be accessed both in Kernel Mode and in User Mode. The corresponding Segment Selector is defined by the `__USER_CS` macro.

- A user data segment shared by all processes in User Mode. The fields of the corresponding Segment Descriptor in the GDT have the following values:
    - Base = 0x00000000
    - Limit = 0xfffff
    - G (granularity flag) = 1, for segment size expressed in pages
    - S (system flag) = 1, for normal code or data segment
    - Type = 2, for data segment that can be read and written
    - DPL (Descriptor Privilege Level) = 3, for User Mode
    - D/B (32-bit address flag) = 1, for 32-bit offset addresses

  This segment overlaps the previous one: they are identical, except for the value of Type. The corresponding Segment Selector is defined by the __USER_DS macro.

- A Task State Segment (TSS) segment for each process. The descriptors of these segments are stored in the GDT. The Base field of the TSS descriptor associated with each process contains the address of the tss field of the corresponding process descriptor. The G flag is cleared, while the Limit field is set to 0xeb, since 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, since processes in User Mode are not allowed to access TSS segments.
- A default LDT segment that is usually shared by all processes. This segment is stored in the default_ldt variable. The default LDT includes a single entry consisting of a null Segment Descriptor. Each process has its own LDT Segment Descriptor, which usually points to the common default LDT segment. The Base field is set to the address of default_ldt and the Limit field is set to 7. If a process requires a real LDT, a new 4096-byte segment is created (it can include up to 511 Segment Descriptors), and the default LDT Segment Descriptor associated with that process is replaced in the GDT with a new descriptor with specific values for the Base and Limit fields.

For each process, therefore, the GDT contains two different Segment Descriptors: one for the TSS segment and one for the LDT segment. The maximum number of entries allowed in the GDT is 12+2xNR_TASKS, where, in turn, NR_TASKS denotes the maximum number of processes. In the previous list we described the six main Segment Descriptors used by Linux. Four additional Segment Descriptors cover Advanced Power Management (APM) features, and four entries of the GDT are left unused, for a grand total of 14.

As we mentioned before, the GDT can have at most 2^{13} = 8192 entries, of which the first is always null. Since 14 are either unused or filled by the system, NR_TASKS cannot be larger than 8180/2 = 4090.

The TSS and LDT descriptors for each process are added to the GDT as the process is created. As we shall see in Section 3.3.2 in Chapter 3, the kernel itself spawns the first process: process running init_task . During kernel initialization, the trap_init( ) function inserts the TSS descriptor of this first process into the GDT using the statement:

```
set_tss_desc(0, &init_task.tss);
```

The first process creates others, so that every subsequent process is the child of some existing process. The copy_thread( ) function, which is invoked from the clone( ) and fork( )

system calls to create new processes, executes the same function in order to set the TSS of the new process:

```
set_tss_desc(nr, &(task[nr]->tss));
```

Since each TSS descriptor refers to a different process, of course, each `Base` field has a different value. The `copy_thread( )` function also invokes the `set_ldt_desc( )` function in order to insert a Segment Descriptor in the GDT relative to the default LDT for the new process.

The kernel data segment includes a process descriptor for each process. Each process descriptor includes its own TSS segment and a pointer to its LDT segment, which is also located inside the kernel data segment.

As stated earlier, the Current Privilege Level of the CPU reflects whether the processor is in User or Kernel Mode and is specified by the `RPL` field of the Segment Selector stored in the `cs` register. Whenever the Current Privilege Level 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.

2.4 Paging in Hardware

The paging unit translates linear addresses into physical ones. It checks 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 6).

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* (they are 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.

In Intel processors, paging is enabled by setting the `PG` flag of the `cr0` register. When `PG = 0`, linear addresses are interpreted as physical addresses.

2.4.1 Regular Paging

Starting with the i80386, 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

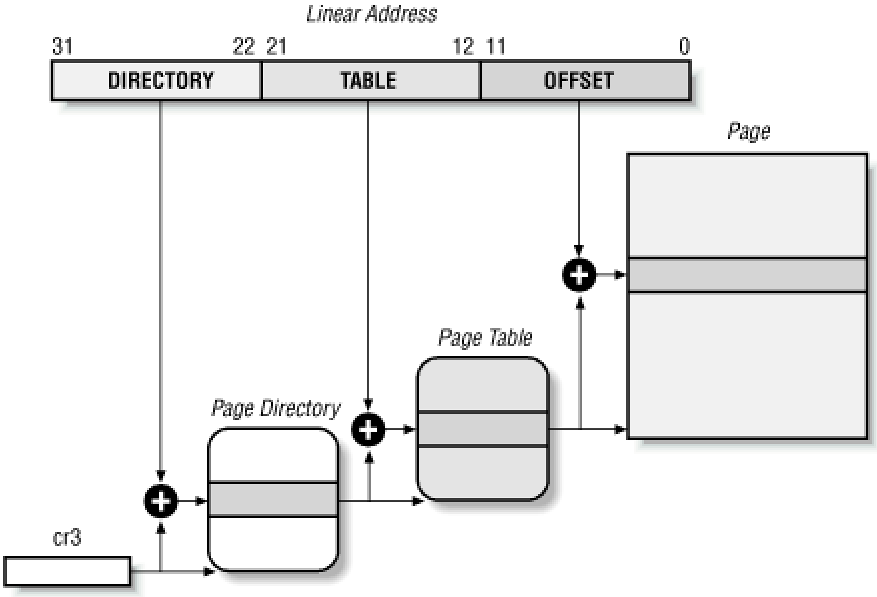
Offset

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 *Page Directory* and the second is called *Page Table*.

The physical address of the Page Directory in use is stored in the `cr3` processor register. 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 2-5). Since it is 12 bits long, each page consists of 4096 bytes of data.

Figure 2-5. Paging by Intel 80x86 processors



Both the Directory and the Table fields are 10 bits long, so Page Directories and Page Tables can include up to 1024 entries. It follows that a Page Directory can address up to 1024 x 1024 x 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 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. (We shall see in Chapter 16, how Linux makes use of this field.)

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

Since 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

Is 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 in the previous case, 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.

`Read/Write` flag

Contains the access right (Read/Write or Read) of the page or of the Page Table (see Section 2.4.3 later in this chapter).

`User/Supervisor` flag

Contains the privilege level required to access the page or Page Table (see Section 2.4.3).

Two flags called `PCD` and `PWT`

Control the way the page or Page Table is handled by the hardware cache (see Section 2.4.6 later in this chapter).

`Page Size` *flag*

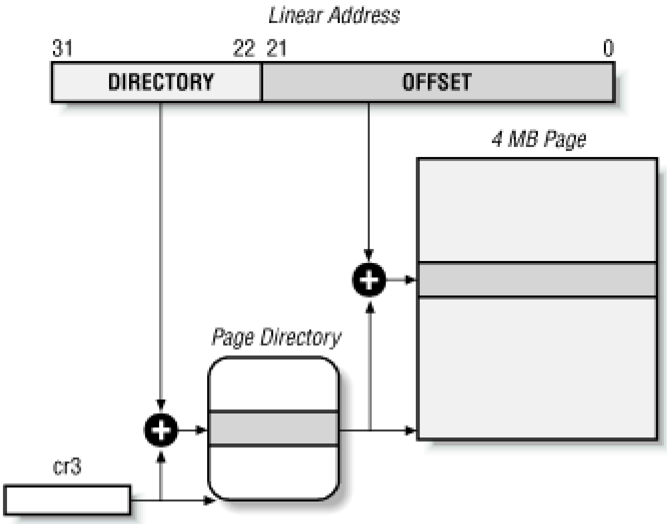
Applies only to Page Directory entries. If it is set, the entry refers to a 4 MB long page frame (see the following section).

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 the `cr2` processor register and generates the exception 14, that is, the "Page fault" exception.

2.4.2 Extended Paging

Starting with the Pentium model, Intel 80x86 microprocessors introduce *extended paging* , which allows page frames to be either 4 KB or 4 MB in size (see Figure 2-6).

Figure 2-6. Extended paging



As we have seen 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 first 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. Extended paging is used to translate large intervals of contiguous linear addresses into corresponding physical ones; in these cases, the kernel can do without intermediate Page Tables and thus save memory.

2.4.3 Hardware Protection Scheme

The paging unit uses a different protection scheme from the segmentation unit. While Intel 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 Section 2.4.1. 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, Execute) associated with segments, only two types of access rights (Read, 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.

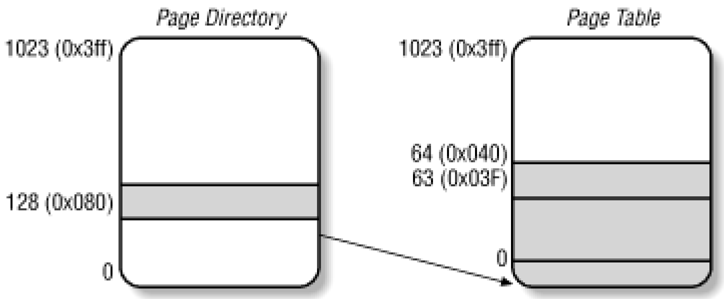
2.4.4 An Example of Paging

A simple example will help in clarifying how paging works.

Let us assume that the kernel has assigned 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 us 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 2-7). If no other linear addresses are assigned to the process, all the remaining 1023 entries of the Page Directory are filled with zeros.

Figure 2-7. An example of paging



The values assumed by the intermediate 10 bits, (that is, the values of the Table field) range from to 0x03f, or from to 63 decimal. Thus, only the first 64 entries of the Page Table are significant. 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 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 since the Page Table entries not assigned to the process are filled with zeros; in particular, their Present flags are all cleared.

2.4.5 Three-Level Paging

Two-level paging is used by 32-bit microprocessors. But in recent years, several microprocessors (such as Compaq's Alpha, and Sun's UltraSPARC) have adopted a 64-bit architecture. In this case, two-level paging is no longer suitable and it is necessary to move up to three-level paging. Let us use a thought experiment to see why.

Start by assuming about as large a page size as is reasonable (since you have to account for pages being transferred routinely to and from disk). Let's choose 16 KB for the page size. Since 1 KB covers a range of 2^{10} addresses, 16 KB covers 2^{14} addresses, so the Offset field would be 14 bits. This leaves 50 bits of the linear address to be distributed between the Table and the Directory fields. If we now decide to reserve 25 bits for each of these two fields, this means that both the Page Directory and the Page Tables of a process would include 2^{25} entries, that is, more than 32 million entries.

Even if RAM is getting cheaper and cheaper, we cannot afford to waste so much memory space just for storing the page tables.

The solution chosen for Compaq's Alpha microprocessors is the following:

- Page frames are 8 KB long, so the Offset field is 13 bits long.
- Only the least significant 43 bits of an address are used. (The most significant 21 bits are always set 0.)
- Three levels of page tables are introduced so that the remaining 30 bits of the address can be split into three 10-bit fields (see Figure 2-9 later in this chapter). So the Page Tables include 2^{10} = 1024 entries as in the two-level paging schema examined previously.

As we shall see in Section 2.5 later in this chapter, Linux's designers decided to implement a paging model inspired by the Alpha architecture.

2.4.6 Hardware Cache

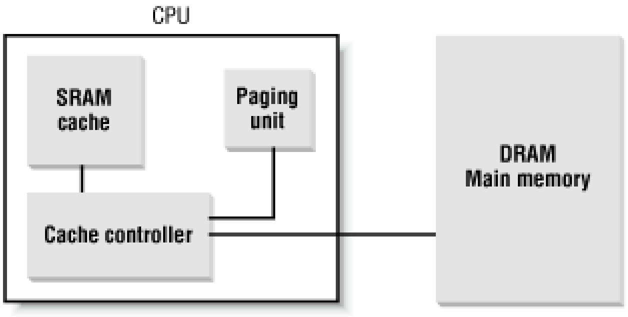
Today's microprocessors have clock rates approaching gigahertz, while dynamic RAM (DRAM) chips have access times in the range of tens of clock cycles. This means that the CPU may be held back considerably while executing instructions that require fetching operands from RAM and/or storing results into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

Hardware cache memories have been 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: 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 thus 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* has been introduced into the Intel 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 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 2-way set of associative cache.

As shown in Figure 2-8, 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 physical address are usually split into three groups: the most significant ones correspond to the tag, the middle ones correspond to the cache controller subset index, the least significant ones to the offset within the line.

Figure 2-8. 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 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 achieves the time saving for which 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. See Section 11.3.2 in Chapter 11.

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 another larger and slower on-chip cache called the *L2-cache*. The consistency between the two 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 that purpose, each Page Directory and each Page Table entry includes two flags: `PCD` specifies whether the cache must be enabled or disabled while accessing data included in the page frame; `PWT` 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.

The `L1_CACHE_BYTES` macro yields the size of a cache line on a Pentium, that is, 32 bytes. In order to optimize the cache hit rate, the kernel adopts the following rules:

- The most frequently used fields of a data structure are placed at the low offset within the data structure so that 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 so that all cache lines are uniformly used.

2.4.7 Translation Lookaside Buffers (TLB)

Besides general-purpose hardware caches, Intel 80x86 processors include other caches called *translation lookaside buffers* or *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.

The `invlpg` instruction can be used to invalidate (that is, to free) a single entry of a TLB. In order to invalidate all TLB entries, the processor can simply write into the `cr3` register that points to the currently used Page Directory.

Since the TLBs serve as caches of page table contents, whenever a Page Table entry is modified, the kernel must invalidate the corresponding TLB entry. To do this, Linux makes use of the `flush_tlb_page(addr)` function, which invokes `__flush_tlb_one( )`. The latter function executes the `invlpg` Assembly instruction:

```
movl $addr,%eax
invlpg (%eax)
```

Sometimes it is necessary to invalidate all TLB entries, such as during kernel initialization. In such cases, the kernel invokes the `__flush_tlb( )` function, which rewrites the current value of `cr3` back into it:

```
movl %cr3, %eax
movl %eax, %cr3
```

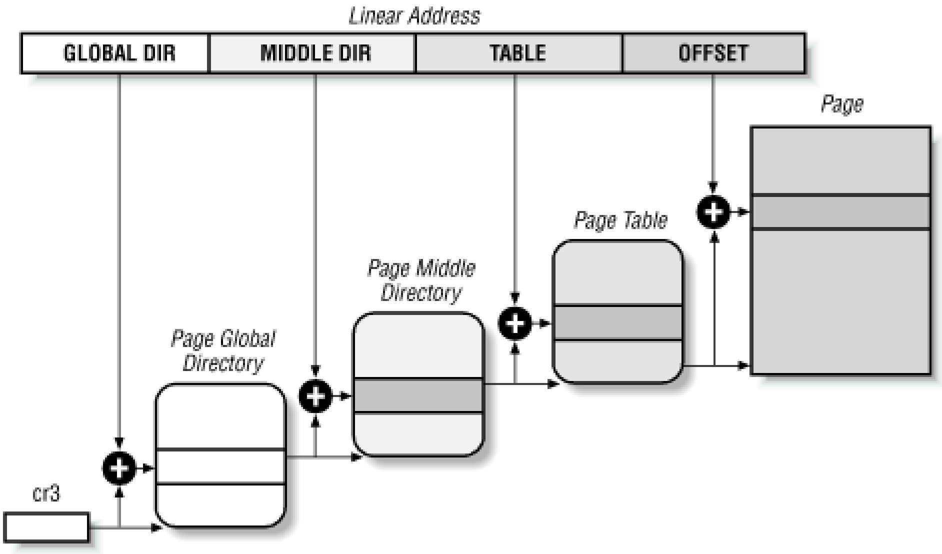
2.5 Paging in Linux

As we explained in Section 2.4.5, Linux adopted a three-level paging model so paging is feasible on 64-bit architectures. Figure 2-9 shows the model, which defines three types of paging tables:

- Page Global Directory
- Page Middle Directory
- Page Table

The Page Global Directory includes 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. The linear address is thus split into four parts. Figure 2-9 does not show the bit numbers because the size of each part depends on the computer architecture.

Figure 2-9. The Linux paging model



Linux 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, thus ensuring an efficient protection against addressing errors.
- Distinguish pages, that is, groups of data, from page frames, that is, 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 16).

As we shall see in Chapter 7, each process has its own Page Global Directory and its own set of Page Tables. When a process switching occurs (see Section 3.2 in Chapter 3), Linux saves in a TSS segment the contents of the `cr3` control register and loads from another TSS segment a new value into `cr3`. Thus, when the new process resumes its execution on the CPU, the paging unit refers to the correct set of page tables.

What happens when this three-level paging model is applied to the Pentium, which uses only two types of page tables? Linux essentially eliminates the Page Middle Directory field by saying that it contains zero bits. However, the position of the Page Middle Directory in the sequence of pointers is kept so that the same code can work on 32-bit and 64-bit architectures. The kernel keeps a position for the Page Middle Directory by setting the number of entries in it to 1 and mapping this single entry into the proper entry of the Page Global Directory.

Mapping logical to linear addresses now becomes a mechanical task, although somewhat complex. The next few sections of this chapter are thus 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 just 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 in subsequent chapters.

2.5.1 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 Pentium processors it yields the value 12. Since all the addresses in a page must fit in the Offset field, the size of a page on Intel 80x86 systems is 2^{12} or the familiar 4096 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 is defined as the value `0xfffff000`; it is used to mask all the bits of the Offset field.

`PMD_SHIFT`

Determines the number of bits in an address that are mapped by the second-level page table. It yields the value 22 (12 from Offset plus 10 from Table). 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. Thus, `PMD_SIZE` yields 2^{22} or 4 MB. The `PMD_MASK` macro yields the value `0xffc00000`; it is used to mask all the bits of the Offset and Table fields.

`PGDIR_SHIFT`

Determines the logarithm of the size of the area a first-level page table can map. Since the Middle Directory field has length 0, this macro yields the same value yielded by `PMD_SHIFT`, which is 22. The `PGDIR_SIZE` macro computes the size of the area mapped by a single entry of the Page Global Directory, that is, of a Page Directory. `PGDIR_SIZE` therefore yields 4 MB. The `PGDIR_MASK` macro yields the value `0xffc00000`, the same as `PMD_MASK`.

`PTRS_PER_PTE` , `PTRS_PER_PMD` , and `PTRS_PER_PGD`

Compute the number of entries in the Page Table, Page Middle Directory, and Page Global Directory; they yield the values 1024, 1, and 1024, respectively.

2.5.2 Page Table Handling

`pte_t`, `pmd_t`, and `pgd_t` are 32-bit data types that describe, respectively, a Page Table, a Page Middle Directory, and a Page Global Directory entry. `pgprot_t` is another 32-bit data type that represents the protection flags associated with a single entry.

Four type-conversion macros (`pte_val( )`, `pmd_val( )`, `pgd_val( )`, and `pgprot_val( )`) cast a 32-bit unsigned integer into the required type. Four other type-conversion macros (`__ pte( )`, `__ pmd( )`, `__ pgd( )`, and `__ pgprot( )`) perform the reverse casting from one of the four previously mentioned specialized types into a 32-bit unsigned integer.

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

- The `pte_none( )`, `pmd_none( )`, and `pgd_none( )` macros yield the value 1 if the corresponding entry has the value 0; otherwise, they yield the value 0.
- The `pte_present( )`, `pmd_present( )`, and `pgd_present( )` macros yield 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 `pte_clear( )`, `pmd_clear( )`, and `pgd_clear( )` macros clear an entry of the corresponding page table.

The macros `pmd_bad( )` and `pgd_bad( )` are used by functions to check Page Global Directory and Page Middle Directory entries passed as input parameters. Each macro 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).

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. Instead, several functions are offered to query the current value of any of the flags included in a Page Table entry:

`pte_read( )`

Returns the value of the `User/Supervisor` flag (indicating whether the page is accessible in User Mode).

`pte_write( )`

Returns 1 if both the `Present` and `Read/Write` flags are set (indicating whether the page is present and writable).

`pte_exec( )`

Returns the value of the `User/Supervisor` flag (indicating whether the page is accessible in User Mode). Notice that pages on the Intel processor cannot be protected against code execution.

`pte_dirty( )`

Returns the value of the `Dirty` flag (indicating whether or not the page has been modified).

`pte_young( )`

Returns the value of the `Accessed` flag (indicating whether the page has been accessed).

Another group of functions sets the value of the flags in a Page Table entry:

`pte_wrprotect( )`

Clears the `Read/Write` flag

`pte_rdprotect` and `pte_exprotect( )`

Clear the `User/Supervisor` flag

`pte_mkwrite( )`

Sets the `Read/Write` flag

`pte_mkread( )` and `pte_mkexec( )`

Set the `User/Supervisor` flag

`pte_mkdirty( )` and `pte_mkclean( )`

Set the `Dirty` flag to 1 and to 0, respectively, thus marking the page as modified or unmodified

`pte_mkyoung( )` and `pte_mkold( )`

Set the `Accessed` flag to 1 and to 0, respectively, thus marking the page as accessed (young) or nonaccessed (old)

`pte_modify(p,v)`

Sets all access rights in a Page Table entry `p` to a specified value `v`

`set_pte`

Writes a specified value into a Page Table entry

Now come the macros that combine a page address and a group of protection flags into a 32-bit page entry or perform the reverse operation of extracting the page address from a page table entry:

`mk_ pte( )`

Combines a linear address and a group of access rights to create a 32-bit Page Table entry.

`mk_ pte_ phys`

Creates a Page Table entry by combining the physical address and the access rights of the page.

`pte_ page( )` and `pmd_ page( )`

Return the linear address of a page from its Page Table entry, and of a Page Table from its Page Middle Directory entry.

`pgd_offset(p,a)`

Receives as parameters a memory descriptor `p` (see Chapter 6) and a linear address `a`. The macro yields the address of the entry in a Page Global Directory that corresponds to the address `a`; the Page Global Directory is found through a pointer within the memory descriptor `p`. The `pgd_offset_k(o)` macro is similar, except that it refers to the memory descriptor used by kernel threads (see Section 3.3.2 in Chapter 3).

`pmd_offset(p,a)`

Receives as parameter a Page Global Directory entry `p` and a linear address `a`; it yields the address of the entry corresponding to the address `a` in the Page Middle Directory referenced by `p`. The `pte_offset(p,a)` macro is similar, but `p` is a Page Middle Directory entry and the macro yields the address of the entry corresponding to `a` in the Page Table referenced by `p`.

The last group of functions of this long and rather boring list were 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 finally add the entry.

Each page table is stored in one page frame; moreover, each process makes use of several page tables. As we shall see in Section 6.1 in Chapter 6, the allocations and deallocations of page frames are expensive operations. Therefore, when the kernel destroys a page table, it adds the corresponding page frame to a software cache. When the kernel must allocate a new page table, it takes a page frame contained in the cache; a new page frame is requested from the memory allocator only when the cache is empty.

The Page Table cache is a simple list of page frames. The `pte_quicklist` macro points to the head of the list, while the first 4 bytes of each page frame in the list are used as a pointer to the next element. The Page Global Directory cache is similar, but the head of the list is yielded by the `pgd_quicklist` macro. Of course, on Intel architecture there is no Page Middle Directory cache.

Since there is no limit on the size of the page table caches, the kernel must implement a mechanism for shrinking them. Therefore, the kernel introduces high and low *watermarks*, which are stored in the `pgt_cache_water` array; the `check_pgt_cache( )` function checks whether the size of each cache is greater than the high watermark and, if so, deallocates page frames until the cache size reaches the low watermark. The `check_ pgt_cache( )` is invoked either when the system is idle or when the kernel releases all page tables of some process.

Now comes the last round of functions and macros:

`pgd_alloc( )`

Allocates a new Page Global Directory by invoking the `get_ pgd_fast( )` function, which takes a page frame from the Page Global Directory cache; if the cache is empty, the page frame is allocated by invoking the `get_ pgd_slow( )` function.

`pmd_alloc(p,a)`

Defined so three-level paging systems can allocate a new Page Middle Directory for the linear address `a`. On Intel 80x86 systems, the function simply returns the input parameter `p`, that is, the address of the entry in the Page Global Directory.

`pte_alloc(p,a)`

Receives as parameters the address of a Page Middle Directory entry `p` and a linear address `a`, and it returns the address of the Page Table entry corresponding to `a`. If the Page Middle Directory entry is null, the function must allocate a new Page Table. To accomplish this, it looks for a free page frame in the Page Table cache by invoking the `get_ pte_fast( )` function. If the cache is empty, the page frame is allocated by invoking `get_ pte_slow( )`. If a new Page Table is allocated, the entry corresponding to `a` is initialized and the `User/Supervisor` flag is set. `pte_alloc_kernel( )` is similar, except that it invokes the `get_ pte_kernel_slow( )` function instead of `get_ pte_slow( )` for allocating a new page frame; the `get_pte_kernel_slow( )` function clears the `User/Supervisor` flag of the new Page Table.

`pte_free( )`, `pte_free_kernel( )`, and `pgd_free( )`

Release a page table and insert the freed page frame in the proper cache. The `pmd_free( )` and `pmd_free_kernel( )` functions do nothing, since Page Middle Directories do not really exist on Intel 80x86 systems.

`free_one_pmd( )`

Invokes `pte_free( )` to release a Page Table.

`free_one_ pgd( )`

Releases all Page Tables of a Page Middle Directory; in the Intel architecture, it just invokes `free_one_ pmd( )` once. Then it releases the Page Middle Directory by invoking `pmd_free( )`.

`SET_PAGE_DIR`

Sets the Page Global Directory of a process. This is accomplished by placing the physical address of the Page Global Directory in a field of the TSS segment of the process; this address is loaded in the `cr3` register every time the process starts or resumes its execution on the CPU. Of course, if the affected process is currently in execution, the macro also directly changes the `cr3` register value so that the change takes effect right away.

`new_ page_tables( )`

Allocates the Page Global Directory and all the Page Tables needed to set up a process address space. It also invokes `SET_PAGE_DIR` in order to assign the new Page Global Directory to the process. This topic will be covered in Chapter 7.

`clear_ page_tables( )`

Clears the contents of the page tables of a process by iteratively invoking `free_one_pgd( )`.

`free_page_tables( )`

Is very similar to `clear_ page_tables( )` , but it also releases the Page Global Directory of the process.

2.5.3 Reserved Page Frames

The kernel's code and data structures are stored in a group of reserved page frames. A page contained in one of these page frames can never be dynamically assigned or swapped to disk.

As a general rule, the Linux kernel is installed in RAM starting from physical address `0x00100000`, that is, from the second megabyte. The total number of page frames required depends on how the kernel has been configured: a typical configuration yields a kernel that can be loaded in less than 2 MBs of RAM.

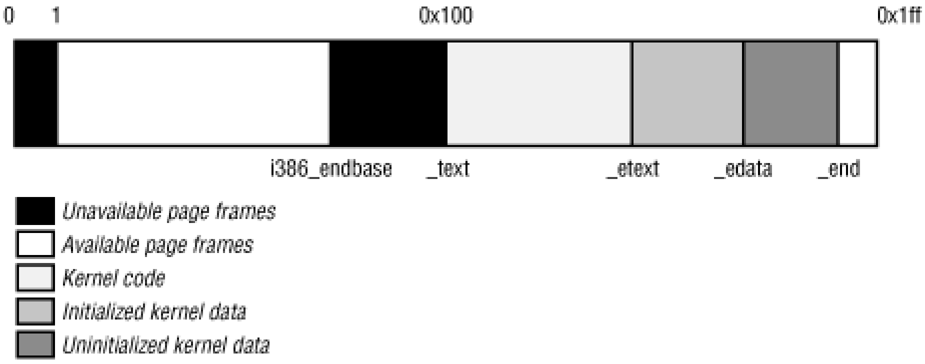
Why isn't the kernel loaded starting with the first available megabyte of RAM? Well, the PC architecture has several peculiarities that must be taken into account:

- Page frame is used by BIOS to store the system hardware configuration detected during the *Power-On Self-Test* (*POST*).
- Physical addresses ranging from `0x000a0000` to `0x000fffff` are reserved to BIOS routines and to map the internal memory of ISA graphics cards (the source of the well-known 640 KB addressing limit in the first MS-DOS systems).
- 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 order 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 2-10 shows how the first 2 MB of RAM are filled by Linux. We have assumed that the kernel requires less than one megabyte of RAM (this is a bit optimistic).

Figure 2-10. The first 512 page frames (2 MB) in Linux 2.2



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.[2]

[2] You can find the linear address of these symbols in the file *System.map*, which is created right after the kernel is compiled.

The linear address corresponding to the first physical address reserved to the BIOS or to a hardware device (usually, `0x0009f000`) is stored in the `i386_endbase` variable. In most cases, this variable is initialized with a value written by the BIOS during the POST phase.

2.5.4 Process Page Tables

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

- Linear addresses from `0x00000000` to `PAGE_OFFSET` -1 can be addressed when the process is in either User or Kernel Mode.
- Linear addresses from `PAGE_OFFSET` to `0xffffffff` can be addressed only when the process is in Kernel Mode.

Usually, the `PAGE_OFFSET` macro yields the value `0xc0000000`: this means that the fourth gigabyte of linear addresses is reserved for the kernel, while the first three gigabytes are accessible from both the kernel and the user programs. However, the value of `PAGE_OFFSET` may be customized by the user when the Linux kernel image is compiled. In fact, as we shall see in the next section, the range of linear addresses reserved for the kernel must include a mapping of all physical RAM installed in the system; moreover, as we shall see in Chapter 7, the kernel also makes use of the linear addresses in this range to remap noncontiguous page frames into contiguous linear addresses. Therefore, if Linux must be installed on a machine having a huge amount of RAM, a different arrangement for the linear addresses might be necessary.

The content of the first entries of the Page Global Directory that map linear addresses lower than `PAGE_OFFSET` (usually the first 768 entries) depends on the specific process. Conversely, the remaining entries are the same for all processes; they are equal to the corresponding entries of the swapper_ pg_dir kernel Page Global Directory (see the following section).

2.5.5 Kernel Page Tables

We now describe how the kernel initializes its own page tables. This is a two-phase activity. In fact, right after the kernel image has been 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 4 MB address space, which is enough for it to install itself in RAM.

In the second phase, the kernel takes advantage of all of the existing RAM and sets up the paging tables properly. The next section examines how this plan is executed.

2.5.5.1 Provisional kernel page tables

Both the Page Global Directory and the Page Table are initialized statically during the kernel compilation. We won't bother mentioning the Page Middle Directories any more since they equate to Page Global Directory entries.

The Page Global Directory is contained in the swapper_ pg_dir variable, while the Page Table that spans the first 4 MB of RAM is contained in the pg0 variable.

The objective of this first phase of paging is to allow these 4 MB to be easily addressed in both real mode and protected mode. Therefore, the kernel must create a mapping from both the linear addresses 0x00000000 through 0x003fffff and the linear addresses PAGE_OFFSET through PAGE_OFFSET+0x3fffff into the physical addresses 0x00000000 through 0x003fffff. In other words, the kernel during its first phase of initialization can address the first 4 MB of RAM (0x00000000 through 0x003fffff) either using linear addresses identical to the physical ones or using 4 MB worth of linear addresses starting from PAGE_OFFSET.

Assuming that PAGE_OFFSET yields the value 0xc0000000, the kernel creates the desired mapping by filling all the swapper_ pg_dir entries with zeros, except for entries and 0x300 (decimal 768); the latter entry spans all linear addresses between 0xc0000000 and 0xc03fffff. The and 0x300 entries are initialized as follows:

- The address field is set to the address of pg0.
- The Present, Read/Write, and User/Supervisor flags are set.
- The Accessed, Dirty, PCD, PWD, and Page Size flags are cleared.

The single pg0 Page Table is also statically initialized, so that the *i* th entry addresses the *i* th page frame.

The paging unit is enabled by the startup_32( ) Assembly-language function. This is achieved by loading in the cr3 control register the address of swapper_pg_dir and by setting the PG flag of the cr0 control register, as shown in the following excerpt:

```
movl $0x101000,%eax
movl %eax,%cr3        /* set the page table pointer.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0        /* ..and set paging (PG) bit */
```

2.5.5.2 Final kernel page table

The final mapping provided by the kernel page tables must transform linear addresses starting from `PAGE_OFFSET` 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 final kernel Page Global Directory is still stored in `swapper_ pg_dir`. It is initialized by the `paging_init( )` function. This function acts on two input parameters:

`start_mem`

The linear address of the first byte of RAM right after the kernel code and data areas.

`end_mem`

The linear address of the end of memory (this address is computed by the BIOS routines during the POST phase).

Linux exploits the extended paging feature of the Pentium processors, enabling 4 MB page frames: it allows a very efficient mapping from `PAGE_OFFSET` into physical addresses by making kernel Page Tables superfluous.[3]

[3] We'll see in Section 6.3 in Chapter 6 that the kernel may set additional mappings for its own use based on 4 KB pages; when this happens, it makes use of Page Tables.

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

```
address = 0;
pg_dir = swapper_pg_dir;
pgd_val(pg_dir[0]) = 0;
pg_dir += (PAGE_OFFSET >> PGDIR_SHIFT);
while (address < end_mem) {
    pgd_val(*pg_dir) = _PAGE_PRESENT+_PAGE_RW+_PAGE_ACCESSED
           +_PAGE_DIRTY +_PAGE_4M+__pa(address);
    pg_dir++;
    address += 0x400000;
}
```

As you can see, the first entry of the Page Global Directory is zeroed out, hence removing the mapping between the first 4 MB of linear and physical addresses. The first Page Table is thus available, so User Mode processes can also use the range of linear addresses between and 4194303.

The User/Supervisor flags in all Page Global Directory entries referencing linear addresses above PAGE_OFFSET are cleared, thus denying to processes in User Mode access to the kernel address space.

The pg0 provisional Page Table is no longer used once swapper_ pg_dir has been initialized.

2.6 Anticipating Linux 2.4

Linux 2.4 introduces two main changes. The TSS Segment Descriptor associated with all existing processes is no longer stored in the Global Descriptor Table. This change removes the hard-coded limit on the number of existing processes. The limit thus becomes the number of available PIDs. In short, you will not find anymore the NR_TASKS macro inside the kernel code, and all data structures whose size was depending on it have been replaced or removed.

The other main change is related to physical memory addressing. Recent Intel 80x86 microprocessors include a feature called Physical Address Extension (PAE), which adds four extra bits to the standard 32-bit physical address. Linux 2.4 takes advantage of PAE and supports up to 64 GB of RAM. However, a linear address is still composed by 32 bits, so that only 4 GB of RAM can be "permanently mapped" and accessed at any time. Linux 2.4 thus recognizes three different portions of RAM: the physical memory suitable for ISA Direct Memory Access (DMA), the physical memory not suitable for ISA DMA but permanently mapped by the kernel, and the "high memory," that is, the physical memory that is not permanently mapped by the kernel.

Chapter 3. Processes

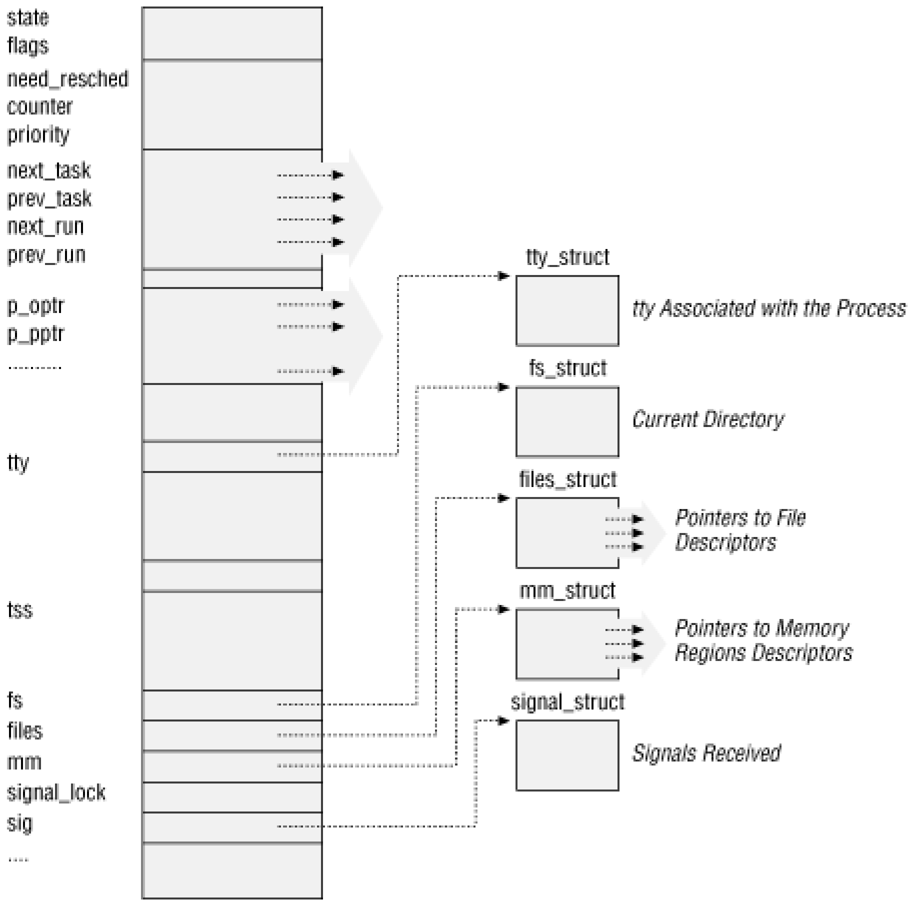
The concept of a *process* is fundamental to any multiprogramming operating system. A process is usually defined as an instance of a program in execution; thus, if 16 users are running `vi` at once, there are 16 separate processes (although they can share the same executable code). Processes are often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

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

3.1 Process Descriptor

In order 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 the CPU or blocked on some 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* , that is, of 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. Not only does it contain many fields itself, but some contain pointers to other data structures that, in turn, contain pointers to other structures. Figure 3-1 describes the Linux process descriptor schematically.

Figure 3-1. The Linux process descriptor



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

3.1.1 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` is set; the remaining flags are cleared. The following are the possible process states:

`TASK_RUNNING`

The process is either executing on the 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, that is, put its state back to `TASK_RUNNING`.

TASK_UNINTERRUPTIBLE

Like the previous state, 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 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. When a process is being monitored by another (such as when a debugger executes a ptrace( ) system call to monitor a test program), any signal may put the process in the TASK_STOPPED state.

TASK_ZOMBIE

Process execution is terminated, but the parent process has not yet issued a wait( )-like system call (wait( ), wait3( ), wait4( ), or waitpid( )) 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 could need it. (See Section 3.4.2 near the end of this chapter.)

3.1.2 Identifying a Process

Although Linux processes can share a large portion of their kernel data structures—an efficiency measure known as *lightweight processes*—each process has its own process descriptor. Each execution context that can be independently scheduled must have its own process descriptor.

Lightweight processes should not be confused with user-mode threads, which are different execution flows handled by a user-level library. For instance, older Linux systems implemented *POSIX threads* entirely in user space by means of the *pthread* library; therefore, a multithreaded program was executed as a single Linux process. Currently, the pthread library, which has been merged into the standard C library, takes advantage of lightweight processes.

The very strict one-to-one correspondence between the process and process descriptor makes the 32-bit process descriptor address[1] a convenient tool 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.

Any Unix-like operating system, on the other hand, allows users to identify processes by means of a number called the *Process ID* (or *PID*). The PID is a 32-bit unsigned integer stored in the pid field of the process descriptor. PIDs are numbered sequentially: the PID of

[1] Technically speaking, these 32 bits are only the offset component of a logical address. However, since Linux makes use of a single kernel data segment, we can consider the offset to be equivalent to a whole logical address. Furthermore, since the base addresses of the code and data segments are set to 0, we can treat the offset as a linear address.

a newly created process is normally the PID of the previously created process incremented by one. However, for compatibility with traditional Unix systems developed for 16-bit hardware platforms, the maximum PID number allowed on Linux is 32767. When the kernel creates the 32768th process in the system, it must start recycling the lower unused PIDs.

At the end of this section, we'll show you how it is possible to derive a process descriptor pointer efficiently from its respective PID. Efficiency is important because many system calls like `kill( )` use the PID to denote the affected process.

3.1.2.1 The task array

Processes are dynamic entities whose lifetimes in the system range from a few milliseconds to months. Thus, the kernel must be able to handle many processes at the same time. In fact, we know from the previous chapter that Linux is able to handle up to `NR_TASKS` processes. The kernel reserves a global static array of size `NR_TASKS` called `task` in its own address space. The elements in the array are process descriptor pointers; a null pointer indicates that a process descriptor hasn't been associated with the array entry.

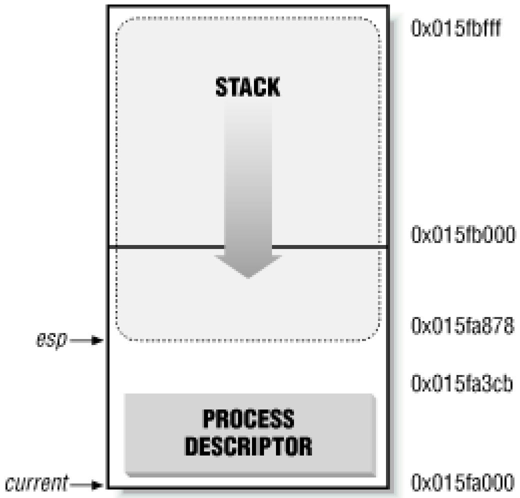
3.1.2.2 Storing a process descriptor

The `task` array contains only pointers to process descriptors, not the sizable descriptors themselves. Since processes are dynamic entities, process descriptors are stored in dynamic memory rather than in the memory area permanently assigned to the kernel. Linux stores two different data structures for each process in a single 8 KB memory area: the process descriptor and the Kernel Mode process stack.

In Section 2.3 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. Since kernel control paths make little use of the stack—even taking into account the interleaved execution of multiple kernel control paths on behalf of the same process—only a few thousand bytes of kernel stack are required. Therefore, 8 KB is ample space for the stack and the process descriptor.

Figure 3-2 shows how the two data structures are stored in the memory area. The process descriptor starts from the beginning of the memory area and the stack from the end.

Figure 3-2. Storing the process descriptor and the process kernel stack



The `esp` register is the CPU stack pointer, which is used to address the stack's top location. On Intel 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 memory area.

The C language allows such a hybrid structure to be conveniently represented by means of the following union construct:

```
union task_union {
    struct task_struct task;
    unsigned long stack[2048];
};
```

After switching from User Mode to Kernel Mode in Figure 3-2, the `esp` register contains the address `0x015fc000`. The process descriptor is stored starting at address `0x015fa000`. The value of the `esp` is decremented as soon as data is written into the stack. Since the process descriptor is less than 1000 bytes long, the kernel stack can expand up to 7200 bytes.

3.1.2.3 The current macro

The pairing between the process descriptor and the Kernel Mode stack just described offers a key benefit in terms of efficiency: the kernel can easily obtain the process descriptor pointer of the process currently running on the CPU from the value of the `esp` register. In fact, since the memory area is 8 KB (2^{13} bytes) long, all the kernel has to do is mask out the 13 least significant bits of `esp` to obtain the base address of the process descriptor. This is done by the `current` macro, which produces some Assembly instructions like the following:

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

After executing these three instructions, the local variable `p` contains the process descriptor pointer of the process running on the CPU.[2]

[2] One drawback to the shared-storage approach is that, for efficiency reasons, 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.

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. Linux 2.0 did not store the kernel stack and the process descriptor together. Instead, it was 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.

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.

A small cache consisting of `EXTRA_TASK_STRUCT` memory areas (where the macro is usually set to 16) is used to avoid unnecessarily invoking the memory allocator. To understand the purpose of this cache, assume for instance that some process is destroyed and that, right afterward, a new process is created. Without the cache, the kernel would have to release an 8 KB memory area to the memory allocator and then, immediately afterward, request another memory area of the same size. This is an example of *memory cache*, a software mechanism introduced to bypass the Kernel Memory Allocator. You will find many other examples of memory caches in the following chapters.

The `task_struct_stack` array contains the pointers to the process descriptors in the cache. Its name comes from the fact that process descriptor releases and requests are implemented respectively as "push" and "pop" operations on the array:

`free_task_struct( )`

This function releases the 8 KB `task_union` memory areas and places them in the cache unless it is full.

`alloc_task_struct( )`

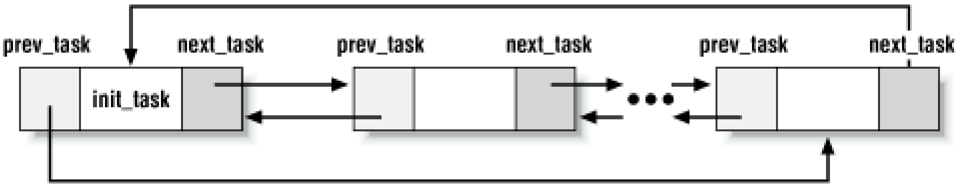
This function allocates 8 KB `task_union` memory areas. The function takes memory areas from the cache if it is at least half-full or if there isn't a free pair of consecutive page frames available.

3.1.2.4 The process list

To allow an efficient search through processes of a given type (for instance, all processes in a runnable state) the kernel creates several lists of processes. Each list consists of pointers to process descriptors. A list pointer (that is, the field that each process uses to point to the next process) is embedded right in the process descriptor's data structure. When you look at the C-language declaration of the `task_struct` structure, the descriptors may seem to turn in on themselves in a complicated recursive manner. However, the concept is no more complicated than any list, which is a data structure containing a pointer to the next instance of itself.

A circular doubly linked list (see Figure 3-3) links together all existing process descriptors; we will call it the *process list*. The `prev_task` and `next_task` fields of each process descriptor are used to implement the list. The head of the list is the `init_task` descriptor referenced by the first element of the `task` array: it is the ancestor of all processes, and it is called *process 0* or *swapper* (see Section 3.3.2 later in this chapter). The `prev_task` field of `init_task` points to the process descriptor inserted last in the list.

Figure 3-3. The process 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 parenthood relationship of the process (see Section 3.1.3 later in this chapter).

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

```
#define for_each_task(p) \
    for (p = &init_task ; (p = p->next_task) != &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).

3.1.2.5 The list of TASK_RUNNING processes

When looking for a new process to run on the CPU, the kernel has to consider only the runnable processes (that is, the processes in the `TASK_RUNNING` state). Since it would be rather inefficient to scan the whole process list, a doubly linked circular list of `TASK_RUNNING` processes called *runqueue* has been introduced. The process descriptors include the `next_run` and `prev_run` fields to implement the runqueue list. As in the previous case, the `init_task` process descriptor plays the role of list header. The `nr_running` variable stores the total number of runnable processes.

The `add_to_runqueue( )` function inserts a process descriptor at the beginning of the list, while `del_from_runqueue( )` removes a process descriptor from the list. For scheduling purposes, two functions, `move_first_runqueue( )` and `move_last_runqueue( )`, are provided to move a process descriptor to the beginning or the end of the runqueue, respectively.

Finally, the `wake_up_process( )` function is used to make a process runnable. It sets the process state to `TASK_RUNNING`, invokes `add_to_runqueue( )` to insert the process in the runqueue list, and increments `nr_running`. It also forces the invocation of the scheduler when the process is either real-time or has a dynamic priority much larger than that of the current process (see Chapter 10).

3.1.2.6 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.

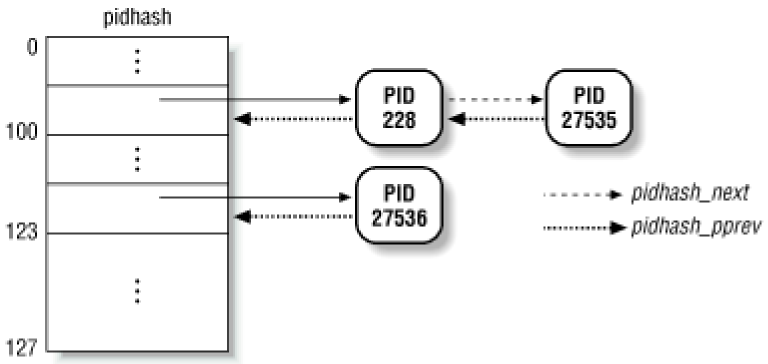
Scanning the process list sequentially and checking the `pid` fields of the process descriptors would be feasible but rather inefficient. In order to speed up the search, a `pidhash` hash table consisting of `PIDHASH_SZ` elements has been introduced (`PIDHASH_SZ` is usually set to `NR_TASKS/4)`. The table entries contain process descriptor pointers. The PID is transformed into a table index using the `pid_hashfn` macro:

```
#define pid_hashfn(x) \
    ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 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 a doubly linked list of colliding process descriptors. These lists are implemented by means of the `pidhash_next` and `pidhash_pprev` fields in the process descriptor. Figure 3-4 illustrates a `pidhash` table with two lists: the processes having PIDs 228 and 27535 hash into the 101st element of the table, while the process having PID 27536 hashes into the 124th element of the table.

Figure 3-4. The pidhash table and chained lists



Hashing with chaining is preferable to a linear transformation from PIDs to table indexes, because a PID can assume any value between and 32767. Since `NR_TASKS`, the maximum number of processes, is usually set to 512, it would be a waste of storage to define a table consisting of 32768 entries.

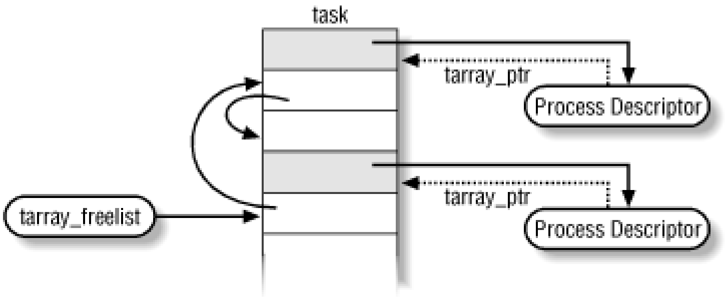
The `hash_ pid( )` and `unhash_ pid( )` functions are invoked to insert and remove a process in the `pidhash` table, respectively. The `find_task_by_pid( )` function searches the hash table and returns the process descriptor pointer of the process with a given PID (or a null pointer if it does not find the process).

3.1.2.7 The list of task free entries

The task array must be updated every time a process is created or destroyed. As with the other lists shown in previous sections, a list is used here to speed additions and deletions. Adding a new entry into the array is done efficiently: instead of searching the array linearly and looking for the first free entry, the kernel maintains a separate doubly linked, noncircular list of free entries. The tarray_freelist variable contains the first element of that list; each free entry in the array points to another free entry, while the last element of the list contains a null pointer. When a process is destroyed, the corresponding element in task is added to the head of the list.

In Figure 3-5, if the first element is counted as 0, the tarray_freelist variable points to element 4 because it is the last freed element. Previously, the processes corresponding to elements 2 and 1 were destroyed, in that order. Element 2 points to another free element of tasks not shown in the figure.

Figure 3-5. An example of task array with free entries



Deleting an entry from the array is also done efficiently. Each process descriptor p includes a tarray_ ptr field that points to the task entry containing the pointer to p.

The get_free_taskslot( ) and add_free_taskslot( ) functions are used to get a free entry and to free an entry, respectively.

3.1.3 Parenthood Relationships Among Processes

Processes created by a program have a parent/child relationship. Since a process can create several children, these have sibling relationships. Several fields must be introduced in a process descriptor to represent these relationships. Processes and 1 are created by the kernel; as we shall see later in the chapter, process 1 (*init*) is the ancestor of all other processes. The descriptor of a process P includes the following fields:

p_opptr (*original 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. Thus, when a shell user starts a background process and exits the shell, the background process becomes the child of *init*.

`p_pptr` (*parent*)

Points to the current parent of P; its value usually coincides with that of `p_opptr`. It may occasionally differ, such as when another process issues a `ptrace( )` system call requesting that it be allowed to monitor P (see Section 19.1.5 in Chapter 19).

`p_cptr` (*child*)

Points to the process descriptor of the youngest child of P, that is, of the process created most recently by it.

`p_ysptr` (*younger sibling*)

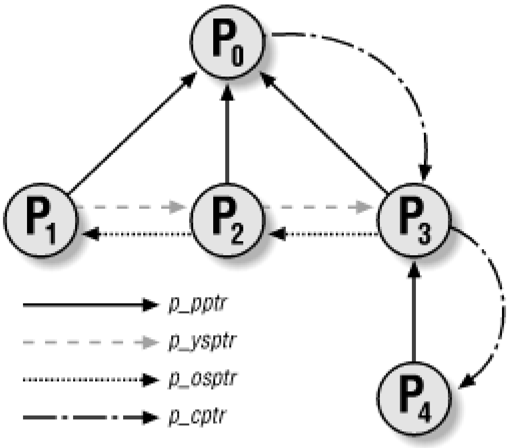
Points to the process descriptor of the process that has been created immediately after P by P's current parent.

`p_osptr` (*older sibling*)

Points to the process descriptor of the process that has been created immediately before P by P's current parent.

Figure 3-6 illustrates the parenthood relationships of a group of processes. Process P0 successively created P1, P2, and P3. Process P3, in turn, created process P4. Starting with `p_cptr` and using the `p_osptr` pointers to siblings, P0 is able to retrieve all its children.

Figure 3-6. Parenthood relationships among five processes



3.1.4 Wait Queues

The runqueue list groups together 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 following choices:

- Processes in a `TASK_STOPPED` or in a `TASK_ZOMBIE` state are not linked in specific lists. There is no need to group them, because either the process PID or the process parenthood relationships may be used by the parent process to retrieve the child process.

- 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 additional lists are called *wait queues*.

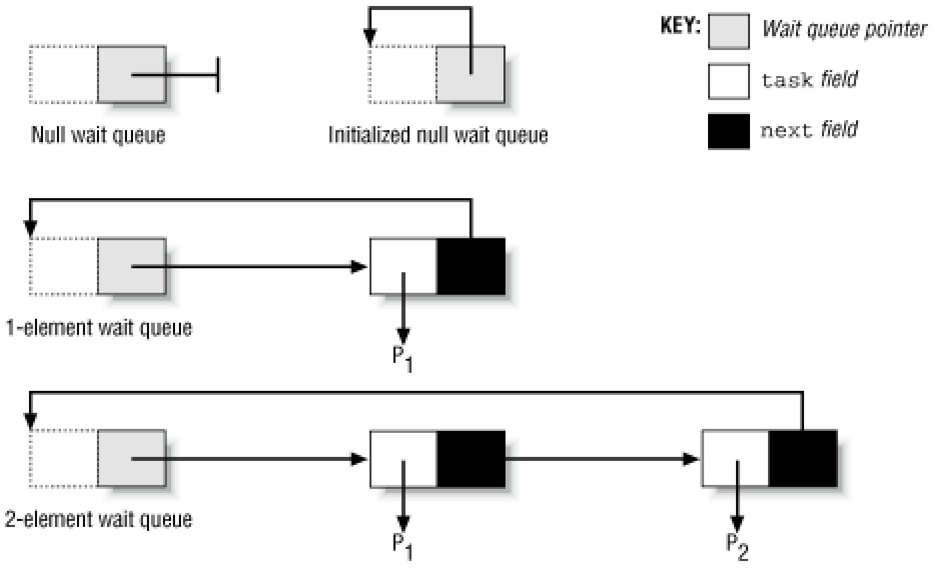
Wait queues have several uses in the kernel, particularly for interrupt handling, process synchronization, and timing. Because these topics are discussed in later chapters, we'll just say here that a process must often wait for some event to occur, such as for a disk operation to terminate, a system 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 awakened by the kernel when some condition becomes true.

Wait queues are implemented as cyclical lists whose elements include pointers to process descriptors. Each element of a wait queue list is of type wait_queue:

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
};
```

Each wait queue is identified by a *wait queue pointer*, which contains either the address of the first element of the list or the null pointer if the list is empty. The next field of the wait_queue data structure points to the next element in the list, except for the last element, whose next field points to a dummy list element. The dummy's next field contains the address of the variable or field that identifies the wait queue minus the size of a pointer (on Intel platforms, the size of the pointer is 4 bytes). Thus, the wait queue list can be considered by kernel functions as a truly circular list, since the last element points to the dummy wait queue structure whose next field coincides with the wait queue pointer (see Figure 3-7).

Figure 3-7. The wait queue data structure



The init_waitqueue( ) function initializes an empty wait queue; it receives the address q of a wait queue pointer as its parameter and sets that pointer to q - 4. The add_wait_queue(q, entry) function inserts a new element with address entry in the wait queue identified by the wait queue pointer q. Since wait queues are modified by interrupt handlers as well as by major kernel functions, the function executes the following operations with disabled interrupts (see Chapter 4):

```
if (*q != NULL)
    entry->next = *q;
else
    entry->next = (struct wait_queue *)(q-1);
*q = entry;
```

Since the wait queue pointer is set to entry, the new element is placed in the first position of the wait queue list. If the wait queue was not empty, the next field of the new element is set to the address of the previous first element. Otherwise, the next field is set to the address of the wait queue pointer minus 4, and thus points to the dummy element.

The remove_wait_queue( ) function removes the element pointed to by entry from a wait queue. Once again, the function must disable interrupts before executing the following operations:

```
next = entry->next;
head = next;
while ((tmp = head->next) != entry)
    head = tmp;
head->next = next;
```

The function scans the circular list to find the element head that precedes entry. It then detaches entry from the list by letting the next field of head point to the element that follows entry. The peculiar format of the wait queue circular list simplifies the code. Moreover, it is very efficient for the following reasons:

- Most wait queues have just one element, which means that the body of the while loop is never executed.
- While scanning the list, there is no need to distinguish the wait queue pointer (the dummy wait queue element) from wait_queue data structures.

A process wishing to wait for a specific condition can invoke any of the following functions:

- The sleep_on( ) function operates on the current process, which we'll call P:

```
void sleep_on(struct wait_queue **p)
{
    struct wait_queue wait;
    current->state = TASK_UNINTERRUPTIBLE;
    wait.task = current;
    add_wait_queue(p, &wait);
    schedule( );
    remove_wait_queue(p, &wait);
}
```

- The function sets P's state to TASK_UNINTERRUPTIBLE and inserts P into the wait queue whose pointer was specified as the parameter. Then it invokes the scheduler, which resumes the execution of another process. When P is awakened, the scheduler resumes execution of the sleep_on( ) function, which removes P from the wait queue.
- The interruptible_sleep_on( ) function is identical to sleep_on( ), except that it sets the state of the current process P to TASK_INTERRUPTIBLE instead of TASK_UNINTERRUPTIBLE so that P can also be awakened 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. In order to do this, they invoke the schedule_timeout( ) function instead of schedule( ) (see Section 5.4.7 in Chapter 5).

Processes inserted in a wait queue enter the TASK_RUNNING state by using either the wake_up or the wake_up_interruptible macros. Both macros use the __wake_up( ) function, which receives as parameters the address q of the wait queue pointer and a bitmask mode specifying one or more states. Processes in the specified states will be woken up; others will be left unchanged. The function essentially executes the following instructions:

```
if (q && (next = *q)) {
    head = (struct wait_queue *)(q-1);
    while (next != head) {
        p = next->task;
        next = next->next;
        if (p->state & mode)
            wake_up_process(p);
    }
}
```

The function checks the state p->state of each process against mode to determine whether the caller wants the process woken up. Only those processes whose state is included in the mode bitmask are actually awakened. The wake_up macro specifies both the TASK_INTERRUPTIBLE and the TASK_UNINTERRUPTIBLE flags in mode, so it wakes up all sleeping processes. Conversely, the wake_up_interruptible macro wakes up only the TASK_INTERRUPTIBLE processes by specifying only that flag in mode. Notice that awakened processes are not removed from the wait queue. A process that has been awakened does not necessarily imply that the wait condition has become true, so the processes could suspend themselves again.

3.1.5 Process Usage Limits

Processes are associated with sets of *usage limits*, which specify the amount of system resources they can use. Specifically, Linux recognizes the following usage limits:

RLIMIT_CPU

Maximum CPU time for the process. 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 9).

RLIMIT_FSIZE

Maximum file size allowed. If the process tries to enlarge a file to a size greater than this value, the kernel sends it a SIGXFSZ signal.

RLIMIT_DATA

Maximum heap size. The kernel checks this value before expanding the heap of the process (see Section 7.6 in Chapter 7).

RLIMIT_STACK

Maximum stack size. The kernel checks this value before expanding the User Mode stack of the process (see Section 7.4 in Chapter 7).

RLIMIT_CORE

Maximum core dump file size. The kernel checks this value when a process is aborted, before creating a core file in the current directory of the process (see Section 9.1.1 in Chapter 9). If the limit is 0, the kernel won't create the file.

RLIMIT_RSS

Maximum number of page frames owned by the process. Actually, the kernel never checks this value, so this usage limit is not implemented.

RLIMIT_NPROC

Maximum number of processes that the user can own (see Section 3.3.1 later in this chapter).

RLIMIT_NOFILE

Maximum number of open files. The kernel checks this value when opening a new file or duplicating a file descriptor (see Chapter 12).

RLIMIT_MEMLOCK

Maximum size of nonswappable memory. The kernel checks this value when the process tries to lock a page frame in memory using the mlock( ) or mlockall( ) system calls (see Section 7.3.4 in Chapter 7).

RLIMIT_AS

Maximum size of process address space. The kernel checks this value when the process uses malloc( ) or a related function to enlarge its address space (see Section 7.1 in Chapter 7).

The usage limits are stored in the rlim field of the process descriptor. The field is an array of elements of type struct rlimit, one for each usage limit:

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

The rlim_cur field is the current usage limit for the resource. For example, current->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 can change the rlim_max field or set the rlim_cur field to a value greater than the corresponding rlim_max field.

Usually, most usage limits contain the value RLIMIT_INFINITY (0x7fffffff), which means that no limit is imposed on the corresponding resource. 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 some 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 system.

3.2 Process Switching

In order 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 is called *process switching* , *task switching*, or *context switching*. The following sections describe the elements of process switching in Linux:

- Hardware context
- Hardware support
- Linux code
- Saving the floating point registers

3.2.1 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, part of the hardware context of a process is stored in the TSS segment, while the remaining part is saved in the Kernel Mode stack. As we learned in Section 2.3 in Chapter 2, the TSS segment coincides with the tss field of the process descriptor.

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 *process switching* as the activity consisting of saving the hardware context of prev and replacing it with the hardware context of next. Since process switches occur quite often, it is important to minimize the time spent in saving and loading hardware contexts.

Earlier versions of Linux took advantage of the hardware support offered by the Intel architecture and performed process switching through a far jmp instruction[3] 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 for the following reasons, Linux 2.2 uses software to perform process switching:

[3] far jmp instructions modify both the cs and eip registers, while simple jmp instructions modify only eip.

- 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 segmentation registers. 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 the current switching code could perhaps be enhanced in the future.

Process switching occurs only in Kernel Mode. The contents of all registers used by a process in User Mode have already been saved 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.

3.2.2 Task State Segment

The Intel 80x86 architecture includes a specific segment type called the *Task State Segment* (TSS), to store hardware contexts. As we saw in Section 2.3 in Chapter 2, each process includes its own TSS segment with a minimum length of 104 bytes. Additional bytes are needed by the operating system to store registers that are not automatically saved by the hardware and to store the *I/O Permission bitmap*. That map is needed because the ioperm( ) and iopl( ) system calls may grant a process in User Mode direct access to specific I/O ports. In particular, if the IOPL field in the eflags register is set to 3, the User Mode process is allowed to access any of the I/O ports whose corresponding bit in the I/O Permission Bit Map is cleared.

The thread_struct structure describes the format of the Linux TSS. An additional area is introduced to store the tr and cr2 registers, the floating point registers, the debug registers, and other miscellaneous information specific to Intel 80x86 processors.

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 whose value cannot be smaller than 0x67 (decimal 103, determined by the minimum TSS segment length mentioned earlier). The S flag of a TSSD is cleared to denote the fact that the corresponding TSS is a *System Segment*.

The `Type` field is set to 11 if the TSSD refers to the TSS of the process currently running on the CPU; otherwise it is set to 9.[4] The second least significant bit of the `Type` field is called the *Busy bit* since it discriminates between the values 9 and 11.[5]

[4] Linux does not make use of a hardware feature that uses the `Type` field in a peculiar way to allow the automatic reexecution of a previously suspended process. Further details may be found in the Pentium manuals.

[5] Since the processor performs a "bus lock" before modifying this bit, a multitasking operating system may test the bit in order to check whether a CPU is trying to switch to a process that's already executing. However, Linux does not make use of this hardware feature (see Chapter 11).

The TSSDs created by Linux are stored in the Global Descriptor Table (GDT), whose base address is stored in the `gdtr` register. The `tr` register contains the TSSD Selector of the process currently running on the CPU. It 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.

As stated earlier, Linux stores part of the hardware context in the `tss` field of the process descriptor. This means that when the kernel creates a new process, it must also initialize the TSSD so that it refers to the `tss` field. Even though the hardware context is saved via software, the TSS segment still plays an important role because it may contain the I/O Permission Bit Map. In fact, 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 IOPL field in the `eflags` register. If it is set to 3 (User Mode process enabled to access I/O ports), it performs the next check; otherwise, it raises a "General protection error" exception.
2. It accesses the `tr` register to determine the current TSS, and thus the proper I/O Permission Bit Map.
3. It checks the bit 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 error" exception.

3.2.3 The switch_to Macro

The `switch_to` macro performs a process switch. It makes use of two parameters denoted as `prev` and `next`: the first is the process descriptor pointer of the process to be suspended, while the second is the process descriptor pointer of the process to be executed on the CPU. The macro is invoked by the `schedule( )` function to schedule a new process on the CPU (see Chapter 10).

The `switch_to` macro is one of the most hardware-dependent routines of the kernel. Here is a description of what it does on an Intel 80x86 microprocessor:

1. Saves the values of `prev` and `next` in the `eax` and `edx` registers, respectively (these values were previously stored in `ebx` and `ecx`):

   ```
   movl %ebx, %eax
   movl %ecx, %edx
   ```

2. Saves the contents of the `esi`, `edi`, 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`:

    ```
    pushl %esi
    pushl %edi
    pushl %ebp
    ```

3. Saves the content of `esp` in `prev->tss.esp` so that the field points to the top of the `prev` Kernel Mode stack:

    ```
    movl %esp, 532(%ebx)
    ```

4. Loads `next->tss.esp` in `esp`. From now on, the kernel operates on the Kernel Mode stack of `next`, so this instruction performs the actual context switch from `prev` to `next`. Since the address of a process descriptor is closely related to that of the Kernel Mode stack (as explained in Section 3.1.2 earlier in this chapter), changing the kernel stack means changing the current process:

    ```
    movl 532(%ecx), %esp
    ```

5. Saves the address labeled `1` (shown later in this section) in `prev->tss.eip`. When the process being replaced resumes its execution, the process will execute the instruction labeled as `1`:

    ```
    movl $1f, 508(%ebx)
    ```

6. On the Kernel Mode stack of `next`, pushes the `next->tss.eip` value, in most cases the address labeled 1:

    ```
    pushl 508(%ecx)
    ```

7. Jumps to the `__switch_to( )` C function:

    ```
    jmp __switch_to
    ```

    This function acts on the `prev` and `next` 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` and `next` parameters from the `eax` and `edx` where we saw earlier they were stored, not from the stack like most functions. To force the function to go to the registers for its parameters, the kernel makes use of `__attribute_ _` and `regparm` keywords, which are nonstandard extensions of the C language implemented by the `gcc` compiler. The `__switch_to( )` function is declared as follows in the *include /asm-i386 /system.h* header file:

    ```
    __switch_to(struct task_struct *prev,
                struct task_struct *next)
       __attribute__(regparm(3))
    ```

    The function completes the process switch started by the `switch_to( )` macro. It includes extended inline Assembly language code that makes for rather complex reading, because the code refers to registers by means of special symbols. In order to simplify the following discussion, we will describe the Assembly language instructions yielded by the compiler:

a. Saves the contents of the `esi` and `ebx` registers in the Kernel Mode stack of `next`, then loads `ecx` and `ebx` with the parameters `prev` and `next`, respectively:

```
pushl %esi
pushl %ebx
movl %eax, %ecx
movl %edx, %ebx
```

b. Executes the code yielded by the `unlazy_fpu( )` macro (see Section 3.2.4 later in this chapter) to optionally save the contents of the mathematical coprocessor registers. As we shall see later, there is no need to load the floating point registers of `next` while performing the context switch:

```
unlazy_fpu(prev);
```

c. Clears the Busy bit (see Section 3.2.2 earlier in this chapter) of `next` and load its TSS selector in the `tr` register:

```
movl 712(%ebx), %eax
andb $0xf8, %al
andl $0xfffffdff, gdt_table+4(%eax)
ltr 712(%ebx)
```

The preceding code is fairly dense. It operates on:

The process's TSSD selector, which is copied from `next->tss.tr` to `eax`.

The 8 least significant bits of the selector, which are stored in `al`.[6] The 3 least significant bits of `al` contain the `RPL` and the `TI` fields of the TSSD.

[6] The `ax` register consists of the 16 least significant bits of `eax`. Moreover, the `al` register consists of the 8 least significant bits of `ax`, while `ah` consists of the 8 most significant bits of `ax`. Similar notations apply to the `ebx`, `ecx`, and `edx` registers. The 13 most significant bits of `ax` specify the TSSD index within the GDT.

Clearing the 3 least significant bits of `al` leaves the TSSD index shifted to the left 3 bits (that is, multiplied by 8). Since the TSSDs are 8 bytes long, the index value multiplied by 8 yields the relative address of the TSSD within the GDT. The `gdt_table+4(%eax)` notation refers to the address of the fifth byte of the TSSD. The `andl` instruction clears the `Busy` bit in the fifth byte, while the `ltr` instruction places the `next->tss.tr` selector in the `tr` register and again sets the `Busy` bit.[7]

[7] Linux must clear the `Busy` bit before loading the value in `tr`, or the control unit will raise an exception.

d. Stores the contents of the `fs` and `gs` segmentation registers in `prev->tss.fs` and `prev->tss.gs`, respectively:

```
movl %fs,564(%ecx)
movl %gs,568(%ecx)
```

e. Loads the `ldtr` register with the `next->tss.ldt` value. This needs to be done only if the Local Descriptor Table used by `prev` differs from the one used by `next`:

```
   movl 920(%ebx),%edx
   movl 920(%ecx),%eax
   movl 112(%eax),%eax
   cmpl %eax,112(%edx)
   je 2f
   lldt 572(%ebx)
2:
```

In practice, the check is made by referring to the `tss.segments` field (at offset 112 in the process descriptor) instead of the `tss.ldt` field.

f. Loads the `cr3` register with the `next->tss.cr3` value. This can be avoided if `prev` and `next` are lightweight processes that share the same Page Global Directory. Since the PGD address of `prev` is never changed, it doesn't need to be saved.

```
movl 504(%ebx),%eax
       cmpl %eax,504(%ecx)
       je 3f
       movl %eax,%cr3
    3:
```

g. Load the `fs` and `gs` segment registers with the values contained in `next->tss.fs` and `next->tss.gs`, respectively. This step logically complements the actions performed in step 7d.

```
movl 564(%ebx),%fs
movl 568(%ebx),%gs
```

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 Section 8.2.6 in Chapter 8).

h. Loads the eight debug registers[8] with the `next->tss.debugreg[i]` values (`0` ≤ `i` ≤ `7`). This is done only if `next` was using the debug registers when it was suspended (that is, field `next->tss.debugreg[7]` is not 0). As we shall see in Chapter 19, these registers are modified only by writing in the TSS, thus there is no need to save them:

```
cmpl $0,760(%ebx)
je 4f
movl 732(%ebx),%esi
movl %esi,%db0
movl 736(%ebx),%esi
movl %esi,%db1
movl 740(%ebx),%esi
```

[8] The Intel 80x86 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 breakpoints, an exception occurs.

```
            movl %esi,%db2
            movl 744(%ebx),%esi
            movl %esi,%db3
            movl 756(%ebx),%esi
            movl %esi,%db6
            movl 760(%ebx),%ebx
            movl %ebx,%db7
        4:
```

i. The function ends up by restoring the original values of the `ebx` and `esi` registers, pushed on the stack in step 7a:

```
popl %ebx
popl %esi
ret
```

When the `ret` instruction is executed, the control unit fetches the value to be loaded in the `eip` program counter from the stack. This value is usually the address of the instruction shown in the following item and labeled `1`, which was stored in the stack by the `switch_to` macro. If, however, `next` was never suspended before because it is being executed for the first time, the function will find the starting address of the `ret_from_fork( )` function (see Section 3.3.1 later in this chapter).

8. The remaining part of the `switch_to` macro includes a few instructions that restore the contents of the `esi`, `edi`, and `ebp` registers. The first of these three instructions is labeled `1`:

```
1:  popl %ebp
    popl %edi
    popl %esi
```

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 second parameter. Therefore, the `esp` register points to the `prev` 's Kernel Mode stack.

3.2.4 Saving the Floating Point Registers

Starting with the Intel 80486, 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. In order to maintain compatibility with older models, however, floating point arithmetic functions are performed by making use of *ESCAPE instructions*, which are instructions with some 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.

Recently, Intel introduced a new set of Assembly 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, since 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.

The Intel 80x86 microprocessors do not automatically save the floating point 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 or an MMX 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 floating point registers only when really needed. To illustrate how it works, let's suppose that a process A is using the mathematical coprocessor. When a context switch occurs, 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 make use of 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 us now describe the data structures introduced to handle selective saving of floating point registers. They are stored in the `tss.i387` subfield, whose format is described by the `i387_hard_struct` structure. The process descriptor also stores the value of two additional flags:

- The `PF_USEDFPU` flag included in the `flags` field. It specifies whether the process used the floating point registers when it was last executing on the CPU.
- The `used_math` field. This flag specifies whether the contents of the `tss.i387` subfield are significant. The flag is cleared (not significant) in two cases:
    - When the process starts executing a new program by invoking an `execve( )` system call (see Chapter 19). Since control will never return to the former program, the data currently stored in `tss.i387` will never be used again.
    - When a process that was executing a program in User Mode starts executing a signal handler procedure (see Chapter 9). Since 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 `tss.i387` before starting the handler and restores them after the handler terminates. Therefore, a signal handler is allowed to make use of the mathematical coprocessor, but it cannot carry on a floating point computation started during the normal program execution flow.

As stated earlier, the `__switch_to( )` function executes the `unlazy_fpu` macro. This macro yields the following code:

```
if (prev->flags & PF_USEDFPU) {
    /* save the floating point registers  */
    asm("fnsave %0" : "=m" (prev->tss.i387));
    /* wait until all data has been transferred */
    asm("fwait");
    prev->flags &= ~PF_USEDFPU;
    /* set the TS flag of cr0 to 1 */
    stts( );
}
```

The `stts( )` macro sets the `TS` flag of `cr0`. In practice, it yields the following Assembly language instructions:

```
movl %cr0, %eax
orb $8, %al
movl %eax, %cr0
```

The contents of the floating point registers are not restored right after a process resumes execution. However, the `TS` flag of `cr0` has been set by `unlazy_fpu( )`. Thus, the first time the process tries to execute an ESCAPE or MMX 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:

```
void math_state_restore(void) {
    asm("clts"); /* clear the TS flag of cr0 */
    if (current->used_math)
        /* load the floating point registers */
        asm("frstor %0": :"m" (current->tss.i387));
    else {
        /* initialize the floating point unit */
        asm("fninit");
        current->used_math = 1;
    }
    current->flags |= PF_USEDFPU;
}
```

Since the process is executing an ESCAPE instruction, this function sets the `PF_USEDFPU` flag. Moreover, the function clears the `TS` flag of `cr0` so that further ESCAPE or MMX instructions executed by the process won't trigger the "Device not available" exception. If the data stored in the `tss.i387` field is valid, the function loads the floating point registers with the proper values. Otherwise, the FPU is reinitialized and all its registers are cleared.

3.3 Creating Processes

Unix operating systems rely heavily on process creation to satisfy user requests. As an example, the shell process 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, and a copy is granted to the child process. This approach makes process creation very slow and inefficient, since it requires copying the entire address space of the parent process. The child process rarely needs to read or modify all the resources already owned by the parent; in many cases, it issues an immediate `execve( )` and wipes out the address space so carefully saved.

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 7.
- *Lightweight processes* allow both the parent and the child to share many per-process kernel data structures, like the paging tables (and therefore the entire User Mode address space) and the open file tables.
- 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.

3.3.1 The clone( ), fork( ), and vfork( ) System Calls

Lightweight processes are created in Linux by using a function named `__clone( )`, which makes use of four 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`

Pointer 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 3 bytes encode a group of clone flags, which specify the resources shared between the parent and the child process. The flags, when set, have the following meanings:

`CLONE_VM`

The memory descriptor and all page tables (see Chapter 7).

`CLONE_FS:`

The table that identifies the root directory and the current working directory.

`CLONE_FILES:`

The table that identifies the open files (see Chapter 12).

`CLONE_SIGHAND:`

The table that identifies the signal handlers (see Chapter 9).

CLONE_PID:

The PID.[9]

[9] As we shall see later, the CLONE_PID flag can be used only by a process having a PID of 0; in a uniprocessor system, no two lightweight processes have the same PID.

CLONE_PTRACE:

If a ptrace( ) system call is causing the parent process to be traced, the child will also be traced.

CLONE_VFORK:

Used for the vfork( ) system call (see later in this section).

child_stack

Specifies the User Mode stack pointer to be assigned to the esp register of the child process. If it is equal to 0, the kernel assigns to the child the current parent stack pointer. Thus, 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. However, this parameter must have a non-null value if the child process shares the same address space as the parent.

__clone( ) is actually a wrapper function defined in the C library (see Section 8.1 in Chapter 8), which in turn makes use of a Linux system call hidden to the programmer, named clone( ). The clone( ) system call receives only the flags and child_stack parameters; the new process always starts its execution from the instruction following the system call invocation. When the system call returns to the __clone( ) function, it determines whether it is in the parent or the child and forces the child to execute the fn( ) function.

The traditional fork( ) system call is implemented by Linux as a clone( ) whose first parameter specifies a SIGCHLD signal and all the clone flags cleared and whose second parameter is 0.

The old vfork( ) system call, described in the previous section, is implemented by Linux as a clone( ) whose first parameter specifies a SIGCHLD signal and the flags CLONE_VM and CLONE_VFORK and whose second parameter is equal to 0.

When either a clone( ), fork( ), or vfork( ) system call is issued, the kernel invokes the do_fork( ) function, which executes the following steps:

1. If the CLONE_PID flag has been specified, the do_fork( ) function checks whether the PID of the parent process is not null; if so, it returns an error code. Only the *swapper* process is allowed to set CLONE_PID; this is required when initializing a multiprocessor system (see Section 11.4.1 in Chapter 11).

2. The `alloc_task_struct( )` function is invoked in order to get a new 8 KB `union task_union` memory area to store the process descriptor and the Kernel Mode stack of the new process.
3. The function follows the `current` pointer to obtain the parent process descriptor and copies it into the new process descriptor in the memory area just allocated.
4. A few checks occur to make sure the user has the resources necessary to start a new process. First, the function checks whether `current->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. 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. The `find_empty_process( )` function is invoked. If the owner of the parent process is not the superuser, this function checks whether `nr_tasks` (the total number of processes in the system) is smaller than `NR_TASKS-MIN_TASKS_LEFT_FOR_ROOT`.[10] If so, `find_empty_process( )` invokes `get_free_taskslot( )` to find a free entry in the `task` array. Otherwise, it returns an error.

[10] A few processes, usually four, are reserved to the superuser; `MIN_TASKS_LEFT_FOR_ROOT` refers to this number. Thus, even if a user is allowed to overload the system with a "fork bomb" (a one-line program that forks itself forever), the superuser can log in, kill some processes, and start searching for the guilty user.

6. The function writes the new process descriptor pointer into the previously obtained `task` entry and sets the `tarray_ptr` field of the process descriptor to the address of that entry (see Section 3.1.2).
7. If the parent process makes use of some kernel modules, the function increments the corresponding reference counters. Each kernel module has its own reference counter, which indicates how many processes are using it. A module cannot be removed unless its reference counter is null (see Appendix B).
8. The function then updates some of the flags included in the `flags` field that have been copied from the parent process:
   a. It clears the `PF_SUPERPRIV` flag, which indicates whether the process has used any of its superuser privileges.
   b. It clears the `PF_USEDFPU` flag.
   c. It clears the `PF_PTRACED` flag unless the `CLONE_PTRACE` parameter flag is set. When set, the `CLONE_PTRACE` flag means that the parent process is being traced with the `ptrace( )` function, so the child should be traced too.
   d. It clears `PF_TRACESYS` flag unless, once again, the `CLONE_PTRACE` parameter flag is set.
   e. It sets the `PF_FORKNOEXEC` flag, which indicates that the child process has not yet issued an `execve( )` system call.
   f. It sets the `PF_VFORK` flag according to the value of the `CLONE_VFORK` flag. This specifies that the parent process must be woken up whenever the process (the child) issues an `execve( )` system call or terminates.
9. Now the function has taken almost everything that it can use from the parent process; the rest of its activities focus on setting up new resources in the child and letting the kernel know that this new process has been born. First, the function invokes the `get_pid( )` function to obtain a new PID, which will be assigned to the child process (unless the `CLONE_PID` flag is set).

10. The function then updates all the process descriptor fields that cannot be inherited from the parent process, such as the fields that specify the process parenthood relationships.
11. Unless specified differently by the `flags` parameter, it invokes `copy_files( )`, `copy_fs( )`, `copy_sighand( )`, and `copy_mm( )` to create new data structures and copy into them the values of the corresponding parent process data structures.
12. It invokes `copy_thread( )` to initialize the Kernel Mode stack of the child process with the values contained in the CPU registers when the `clone( )` call was issued (these values have been saved in the Kernel Mode stack of the parent, as described in Chapter 8). However, the function forces the value into the field corresponding to the `eax` register. The `tss.esp` field of the TSS of the child process is initialized with the base address of the Kernel Mode stack, and the address of an Assembly language function (`ret_from_fork( )`) is stored in the `tss.eip` field.
13. It uses the `SET_LINKS` macro to insert the new process descriptor in the process list.
14. It uses the `hash_pid( )` function to insert the new process descriptor in the `pidhash` hash table.
15. It increments the values of `nr_tasks` and `current->user->count`.
16. It sets the `state` field of the child process descriptor to `TASK_RUNNING` and then invokes `wake_up_process( )` to insert the child in the runqueue list.
17. If the `CLONE_VFORK` flag has been specified, the function suspends the parent process until the child releases its memory address space (that is, until the child either terminates or executes a new program). In order to do this, the process descriptor includes a kernel semaphore called `vfork_sem` (see Section 11.2.4 in Chapter 11).
18. It returns the PID of the child, which will be eventually be read by the parent process in User Mode.

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 will bestow this favor on the child process by loading a few CPU registers with the values of the `tss` field of the child's process descriptor. In particular, `esp` will be loaded with `tss.esp` (that is, with the address of child's Kernel Mode stack), and `eip` will be loaded with the address of `ret_from_fork( )`. This Assembly language function, in turn, invokes the `ret_from_sys_call( )` function (see Chapter 8), which reloads all other registers with the values stored in the stack and forces the CPU back to User Mode. The new process will then start 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 for the child and equal to the PID for the child's parent.

The child process will execute the same code as the parent, except that the fork will return a null PID. The developer of the application can exploit this fact, in the 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.

3.3.2 Kernel Threads

Traditional Unix systems delegate some critical tasks to intermittently running processes, including flushing disk caches, swapping out unused page frames, 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. Since 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:

- Each kernel thread executes a single specific kernel function, while regular processes execute kernel functions only through system calls.
- Kernel threads run only in Kernel Mode, while regular processes run alternatively in Kernel Mode and in User Mode.
- Since kernel threads run only in Kernel Mode, they use only linear addresses greater than `PAGE_OFFSET`. Regular processes, on the other hand, use all 4 gigabytes of linear addresses, either in User Mode or in Kernel Mode.

3.3.2.1 Creating a kernel thread

The `kernel_thread( )` function creates a new kernel thread and can be executed only by another kernel thread. The function contains mostly inline Assembly language code, but it is somewhat equivalent to the following:

```
int kernel_thread(int (*fn)(void *), void * arg,
                  unsigned long flags)
{
    pid_t p;
    p = clone( 0, flags | CLONE_VM );
    if ( p )        /* parent */
        return p;
    else {          /* child */
        fn(arg);
        exit(  );
   }
}
```

3.3.2.2 Process 0

The ancestor of all processes, called *process 0* or, for historical reasons, the *swapper process*, is a kernel thread created from scratch during the initialization phase of Linux by the `start_kernel( )` function (see Appendix A). This ancestor process makes use of the following data structures:

- A process descriptor and a Kernel Mode stack stored in the `init_task_union` variable. The `init_task` and `init_stack` macros yield the addresses of the process descriptor and the stack, respectively.
- The following tables, which the process descriptor points to:
    - `init_mm`
    - `init_mmap`
    - `init_fs`
    - `init_files`
    - `init_signals`

  The tables are initialized, respectively, by the following macros:

    - `INIT_MM`
    - `INIT_MMAP`
    - `INIT_FS`
    - `INIT_FILES`

    - `INIT_SIGNALS`
- A TSS segment, initialized by the `INIT_TSS` macro.
- Two Segment Descriptors, namely a TSSD and an LDTD, which are stored in the GDT.
- A Page Global Directory stored in `swapper_pg_dir`, which may be considered as the kernel Page Global Directory since it is used by all kernel threads.

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_FILES | CLONE_SIGHAND);
```

The newly created kernel thread has PID 1 and shares with process all per-process kernel data structures. Moreover, when selected from the scheduler, the *init* process starts executing the `init( )` function.

After having created the *init* process, process executes the `cpu_idle( )` function, which essentially consists of repeatedly executing the `hlt` Assembly language instruction with the interrupts enabled (see Chapter 4). Process is selected by the scheduler only when there are no other processes in the `TASK_RUNNING` state.

3.3.2.3 Process 1

The kernel thread created by process executes the `init( )` function, which in turn invokes the `kernel_thread( )` function four times to initiate four kernel threads needed for routine kernel tasks:

```
kernel_thread(bdflush, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kupdate, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kpiod, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kswapd, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
```

As a result, four additional kernel threads are created to handle the memory cache and the swapping activity:

kflushd (also *bdflush*)

Flushes "dirty" buffers to disk to reclaim memory, as described in Section 14.1.5 in Chapter 14

kupdate

Flushes old "dirty" buffers to disk to reduce risks of filesystem inconsistencies, as described in Section 14.1.5 in Chapter 14

kpiod

Swaps out pages belonging to shared memory mappings, as described in Section 16.5.2 in Chapter 16,

kswapd

Performs memory reclaiming, as described in Section 16.7.6 in Chapter 16

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. The *init* process never terminates, since it creates and monitors the activity of all the processes that implement the outer layers of the operating system.

3.4 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( )` system call. This system call may be inserted by the programmer explicitly. Additionally, the `exit( )` system call is always executed when the control flow reaches the last statement of the main procedure (the `main( )` function in C programs).

Alternatively, the kernel may force a process to die. This typically occurs when the process has received a signal that it cannot handle or ignore (see Chapter 9) 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).

3.4.1 Process Termination

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 executes the following actions:

1. Sets the `PF_EXITING` flag in the `flag` field of the process descriptor to denote that the process is being eliminated.
2. Removes, if necessary, the process descriptor from an IPC semaphore queue via the `sem_exit( )` function (see Chapter 18) or from a dynamic timer queue via the `del_timer( )` function (see Chapter 5).
3. Examines the process's data structures related to paging, filesystem, open file descriptors, and signal handling, respectively, with the `__exit_mm( )`, `__exit_files( )`, `__exit_fs( )`, and _ `_exit_sighand( )` functions. These functions also remove any of these data structures if no other process is sharing it.
4. Sets the `state` field of the process descriptor to `TASK_ZOMBIE`. We shall see what happens to zombie processes in the following section.
5. Sets the `exit_code` field of the process descriptor to the process termination code. This value is either the `exit( )` system call parameter (normal termination), or an error code supplied by the kernel (abnormal termination).

6. Invokes the `exit_notify( )` function to update the parenthood relationships of both the parent process and the children processes. All children processes created by the terminating process become children of the *init* process.
7. Invokes the `schedule( )` function (see Chapter 10) to select a new process to run. Since a process in a `TASK_ZOMBIE` state is ignored by the scheduler, the process will stop executing right after the `switch_to` macro in `schedule( )` is invoked.

3.4.2 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 for any of its children. A process may, for instance, create a child process to perform a specific task and then invoke a `wait( )`-like system call 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.

In order 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 `TASK_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 might be flooded with zombie processes that might end up using all the available `task` entries. 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( )` function releases the process descriptor of a zombie process by executing the following steps:

1. Invokes the `free_uid( )` function to decrement by 1 the number of processes created up to now by the user owner of the terminated process. This value is stored in the `user_struct` structure mentioned earlier in the chapter.
2. Invokes `add_free_taskslot( )` to free the entry in `task` that points to the process descriptor to be released.
3. Decrements the value of the `nr_tasks` variable.
4. Invokes `unhash_pid( )` to remove the process descriptor from the `pidhash` hash table.
5. Uses the `REMOVE_LINKS` macro to unlink the process descriptor from the process list.
6. Invokes the `free_task_struct( )` function to release the 8 KB memory area used to contain the process descriptor and the Kernel Mode stack.

3.5 Anticipating Linux 2.4

The new kernel supports a huge number of users and groups, because it makes use of 32-bit UIDs and GIDs.

In order to raise the hardcoded limit on the number of processes, Linux 2.4 removes the `tasks` array, which previously included pointers to all process descriptors.

Moreover, Linux 2.4 no longer includes a Task State Segment for each process. The `tss` field in the process descriptor has thus been replaced by a pointer to a data structure storing the information that was previously in the TSS, namely the register contents and the I/O bitmap. Linux 2.4 makes use of just one TSS for each CPU in the system. When a context switch occurs, the kernel uses the per-process data structures to save and restore the register contents and to fill the I/O bitmap in the TSS of the executing CPU.

Linux 2.4 enhances wait queues. Sleeping processes are now stored in lists implemented through the efficient `list_head` data type. Moreover, the kernel is now able to wake up just a single process that is sleeping in a wait queue, thus greatly improving the efficiency of semaphores.

Finally, Linux 2.4 adds a new flag to the `clone( )` system call: `CLONE_PARENT` allows the new lightweight process to have the same parent as the process that invoked the system call.

Chapter 6. Memory Management

We saw in Chapter 2, how Linux takes advantage of Intel's segmentation and paging circuits to translate logical addresses into physical ones. In the same chapter, we 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.

This chapter, which consists of three main sections, describes how the kernel allocates dynamic memory for its own use. Section 6.1 and Section 6.2 illustrate two different techniques for handling physically contiguous memory areas, while Section 6.3 illustrates a third technique that handles noncontiguous memory areas.

6.1 Page Frame Management

We saw in Section 2.4 in Chapter 2 how the Intel Pentium processor can use two different page frame sizes: 4 KB and 4 MB. Linux adopts the smaller 4 KB page frame size as the standard memory allocation unit. This makes things simpler for two reasons:

- The paging circuitry automatically checks whether the page being addressed is contained in some page frame; furthermore, each page frame is hardware-protected through the flags included in the Page Table entry that points to it. By choosing a 4 KB allocation unit, the kernel can directly determine the memory allocation unit associated with the page where a page fault exception occurs.
- The 4 KB size is a multiple of most disk block sizes, so transfers of data between main memory and disks are more efficient. Yet this smaller size is much more manageable than the 4 MB size.

The kernel must keep track of the current status of each page frame. For instance, it must be able to distinguish the page frames used to contain pages belonging 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 or not. This sort of state information is kept in an array of descriptors, one for each page frame. The descriptors of type `struct page` have the following format:

```
typedef struct page {
    struct page *next;
    struct page *prev;
    struct inode *inode;
    unsigned long offset;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct wait_queue *wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
} mem_map_t;
```

We shall describe only a few fields (the remaining ones will be discussed in later chapters dealing with filesystems, I/O buffers, memory mapping, and so on):

`count`

Set 0 to if the corresponding page frame is free; set to a value greater than if the page frame has been assigned to one or more processes or if it is used for some kernel data structures.

`prev , next`

Used to insert the descriptor in a doubly linked circular list. The meaning of these fields depends on the current use of the page frame.

`flags`

An array of up to 32 flags (see Table 6-1) describing the status of the page frame. For each `PG_`*xyz* flag, a corresponding `Page`*Xyz* macro has been defined to read or set its value.

Some of the flags listed in Table 6-1 are explained in later chapters. The `PG_DMA` flag exists because of a limitation on Direct Memory Access (DMA) processors for ISA buses: such DMA processors are able to address only the first 16 MB of RAM, hence page frames are divided into two groups depending on whether they can be addressed by the DMA or not. (Section 13.1.4 in Chapter 13, gives further details on DMAs.) In this chapter, the term "DMA" will always refer to DMA for ISA buses.

Table 6-1. Flags Describing the Status of a Page Frame

Flag Name	Meaning
`PG_decr_after`	See Section 16.4.3 in Chapter 16.
`PG_dirty`	Not used.
`PG_error`	An I/O error occurred while transferring the page.
`PG_free_after`	See Section 15.1.1 in Chapter 15.
`PG_DMA`	Usable by ISA DMA (see text).
`PG_locked`	Page cannot be swapped out.
`PG_referenced`	Page frame has been accessed through the hash table of the page cache (see Section 14.2 in Chapter 14).
`PG_reserved`	Page frame reserved to kernel code or unusable.
`PG_skip`	Used on SPARC/SPARC64 architectures to "skip" some parts of the address space.
`PG_Slab`	Included in a slab: see Section 6.2 later in this chapter.
`PG_swap_cache`	Included in the swap cache; see Section 16.3 in Chapter 16
`PG_swap_unlock_after`	See Section 16.4.3 in Chapter 16.
`PG_uptodate`	Set after completing a read operation, unless a disk I/O error happened.

All the page frame descriptors on the system are included in an array called `mem_map`. Since each descriptor is less than 64 bytes long, `mem_map` requires about four page frames for each megabyte of RAM. The `MAP_NR` macro computes the number of the page frame whose address is passed as a parameter, and thus the index of the corresponding descriptor in `mem_map`:

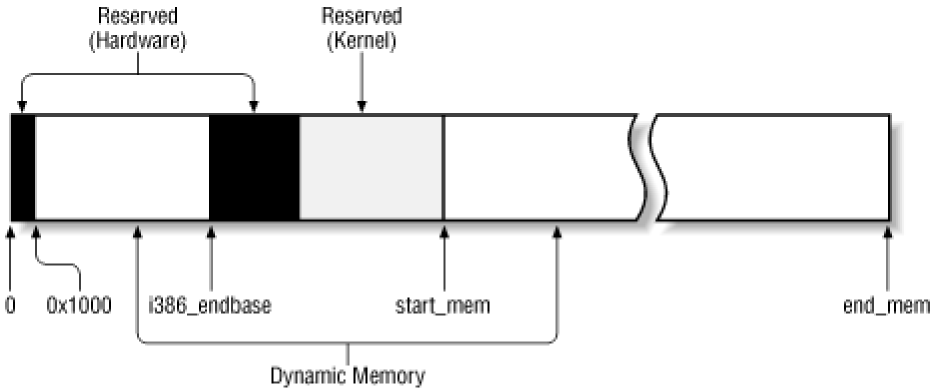
```
#define MAP_NR(addr)    (__pa(addr) >> PAGE_SHIFT)
```

The macro makes use of the __ pa macro, which converts a logical address to a physical one.

Dynamic memory, and the values used to refer to it, are illustrated in Figure 6-1. Page frame descriptors are initialized by the `free_area_init( )` function, which acts on two parameters: `start_mem` denotes the first linear address of the dynamic memory immediately after the kernel memory, while `end_mem` denotes the last linear address of the dynamic memory plus 1 (see Section 2.5.3 and Section 2.5.5 in Chapter 2). The `free_area_init( )` function also considers the `i386_endbase` variable, which stores the initial address of the reserved page frames. The function allocates a suitably sized memory area to `mem_map`. The function then initializes the area by setting all fields to 0, except for the `flags` fields, in which it sets the `PG_DMA` and `PG_reserved` flags:

```
mem_map = (mem_map_t *) start_mem;
p = mem_map + MAP_NR(end_mem);
start_mem = ((unsigned long) p + sizeof(long) - 1) &
                ~(sizeof(long)-1);
memset(mem_map, 0, start_mem - (unsigned long) mem_map);
do {
    --p;
    p->count = 0;
    p->flags = (1 << PG_DMA) | (1 << PG_reserved);
} while (p > mem_map);
```

Figure 6-1. Memory layout



Subsequently, the `mem_init( )` function clears both the `PG_reserved` flag of the page frames, so they can be used as dynamic memory (see Section 2.5.3 in Chapter 2), and the `PG_DMA` flags of all page frames having physical addresses greater than or equal to `0x1000000`. This is done by the following fragment of code:

```
start_low_mem = PAGE_SIZE + PAGE_OFFSET;
num_physpages = MAP_NR(end_mem);
while (start_low_mem < i386_endbase) {
    clear_bit(PG_reserved,
              &mem_map[MAP_NR(start_low_mem)].flags);
    start_low_mem += PAGE_SIZE;
}
```

```
while (start_mem < end_mem) {
    clear_bit(PG_reserved,
              &mem_map[MAP_NR(start_mem)].flags);
    start_mem += PAGE_SIZE;
}
for (tmp = PAGE_OFFSET ; tmp < end_mem ; tmp += PAGE_SIZE) {
    if (tmp >= PAGE_OFFSET+0x1000000)
        clear_bit(PG_DMA, &mem_map[MAP_NR(tmp)].flags);
    if (PageReserved(mem_map+MAP_NR(tmp))) {
        if (tmp >= (unsigned long) &_text
                && tmp < (unsigned long) &_edata)
            if (tmp < (unsigned long) &_etext)
                codepages++;
            else
                datapages++;
        else if (tmp >= (unsigned long) &__init_begin
                   && tmp < (unsigned long) &__init_end)
            initpages++;
        else if (tmp >= (unsigned long) &__bss_start
                 && tmp < (unsigned long) start_mem)
            datapages++;
        else
            reservedpages++;
        continue;
    }
    mem_map[MAP_NR(tmp)].count = 1;
    free_page(tmp);
}
```

First, the `mem_init( )` function determines the value of `num_physpages`, the total number of page frames present in the system. It then counts the number of page frames of type `PG_reserved`. Several symbols produced while compiling the kernel (we described some of them in Section 2.5.3 in Chapter 2) enable the function to count the number of page frames reserved for the hardware, kernel code, and kernel data and the number of page frames used during kernel initialization that can be successively released.

Finally, `mem_init( )` sets the `count` field of each page frame descriptor associated with the dynamic memory to 1 and calls the `free_ page( )` function (see Section 6.1.2 later in this chapter). Since this function increments the value of the variable `nr_free_pages`, that variable will contain the total number of page frames in the dynamic memory at the end of the loop.

6.1.1 Requesting and Releasing Page Frames

After having seen how the kernel allocates and initializes the data structures for page frame handling, we now look at how page frames are allocated and released. Page frames can be requested by making use of four slightly differing functions and macros:

`__get_free_pages(gfp_mask, order)`

Function used to request 2^{order} contiguous page frames.

`__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)
```

__get_free_page(gfp_mask)

Macro used to get a single page frame; it expands to:

```
__get_free_pages(gfp_mask, 0)
```

get_free_page(gfp_mask) :

Function that invokes:

```
__get_free_page(gfp_mask)
```

and then fills the page frame obtained with zeros.

The parameter gfp_mask specifies how to look for free page frames. It consists of the following flags:

__GFP_WAIT

Set if the kernel is allowed to discard the contents of page frames in order to free memory before satisfying the request.

__GFP_IO

Set if the kernel is allowed to write pages to disk in order to free the corresponding page frames. (Since swapping can block the process in Kernel Mode, this flag must be cleared when handling interrupts or modifying critical kernel data structures.)

__GFP_DMA

Set if the requested page frames must be suitable for DMA. (The hardware limitation that gives rise to this flag was explained in Section 6.1.)

__GFP_HIGH , __GFP_MED , __GFP_LOW

Specify the request priority. _ _GFP_LOW is usually associated with dynamic memory requests issued by User Mode processes, while the other priorities are associated with kernel requests.

In practice, Linux uses the predefined combinations of flag values shown in Table 6-2; the group name is what you'll encounter in the source code.

Table 6-2. Groups of Flag Values Used to Request Page Frames

Group Name	__GFP_WAIT	__GFP_IO	Priority
GFP_ATOMIC	0	0	__GFP_HIGH
GFP_BUFFER	1	0	__GFP_LOW
GFP_KERNEL	1	1	__GFP_MED
GFP_NFS	1	1	__GFP_HIGH
GFP_USER	1	1	__GFP_LOW

Page frames can be released through any of the following three functions and macros:

`free_pages(addr, order)`

This function checks the page descriptor of the page frame having physical address `addr`; if the page frame is not reserved (i.e., if the `PG_reserved` flag is equal to 0), it decrements the `count` field of the descriptor. If `count` becomes 0, it assumes that 2^{order} contiguous page frames starting from `addr` are no longer used. In that case, the function invokes `free_ pages_ok( )` to insert the page frame descriptor of the first free page in the proper list of free page frames (described in the following section).

`__free_page(p)`

Similar to the previous function, except that it releases the page frame whose descriptor is pointed to by parameter `p`.

`free_page(addr)`

Macro used to release the page frame having physical address `addr`; it expands

to `free_pages(addr,0)` .

6.1.2 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:

- Make use of 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 in order to satisfy a request for a smaller one.

The second approach is the one preferred by the kernel for two good reasons:

- In some cases, contiguous page frames are really necessary, since 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). Since the DMA ignores the paging circuitry and accesses 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, since they make the CPU flush the contents of the translation lookaside buffers.

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 10 lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 contiguous page frames, respectively. 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 x 2^{12}.

We'll show how the algorithm works through a simple example.

Assume there is a request for a group of 128 contiguous page frames (i.e., a half-megabyte). The algorithm checks first whether a free block in the 128-page-frame list exists. If there is no such block, the algorithm looks for the next larger block, that is, a free block in the 256-page-frame list. If such a block exists, the kernel allocates 128 of the 256 page frames to satisfy the request and inserts the remaining 128 page frames into the list of free 128-page-frame blocks. If there is no free 256-page block, it then looks for the next larger block, that is, a free 512-page-frame block. If such a block exists, it allocates 128 of the 512 page frames to satisfy the request, inserts the first 256 of the remaining 384 page frames into the list of free 256-page-frame blocks, and inserts the last 128 of the remaining 384 page frames into the list of free 128-page-frame blocks. If the list of 512-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 together pairs of free buddy blocks of size *b* into a single block of size 2*b*. Two blocks are considered buddy 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 x *b* x 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.

6.1.2.1 Data structures

Linux makes use of two different buddy systems: one handles the page frames suitable for ISA DMA, while the other one handles the remaining page frames. Each buddy system relies on the following main data structures:

- The `mem_map` array introduced previously.
- An array having 10 elements of type `free_area_struct`, one element for each group size. The variable `free_area[0]` points to the array used by the buddy system for the page frames that are not suitable for ISA DMA, while `free_area[1]` points to the array used by the buddy system for page frames suitable for ISA DMA.
- Ten binary arrays named *bitmaps*, one for each group size. Each buddy system has its own set of bitmaps, which it uses to keep track of the blocks it allocates.

Each element of the `free_area[0]` and `free_area[1]` arrays is a structure of type `free_area_struct`, which is defined as follows:

```
struct free_area_struct {
    struct page *next;
    struct page *prev;
    unsigned int *map;
    unsigned long count;
};
```

Notice that the first two fields of this structure match the corresponding fields of a page descriptor; in fact, pointers to `free_area_struct` structures are sometimes used as pointers to page descriptors.

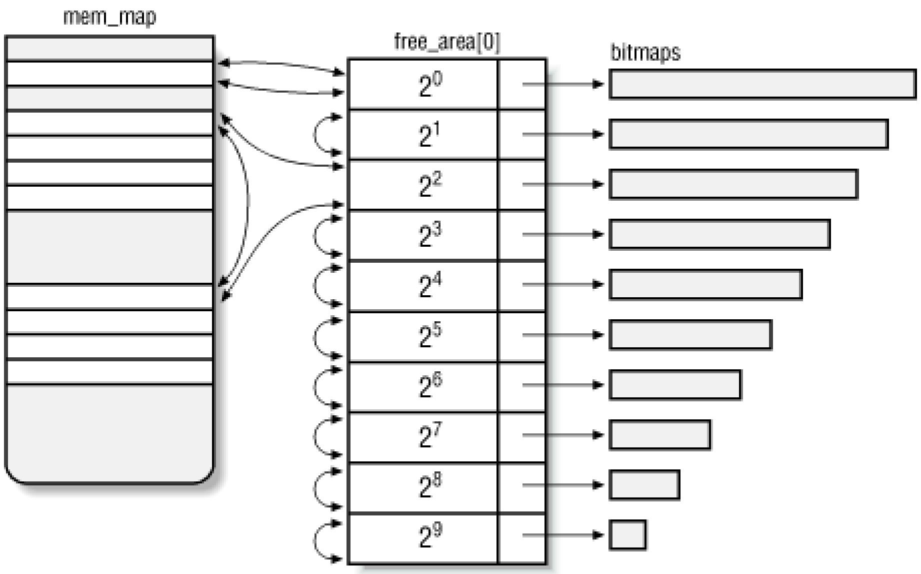
The *k* th element of either the `free_area[0]` or the `free_area[1]` array is associated with a doubly linked circular list of blocks of size 2^k, implemented through the `next` and `prev` fields. Each member of such a list is the descriptor of the first page frame of a block. The `count` field of each `free_area_struct` structure stores the number of elements in the corresponding list.

The `map` field points to a bitmap whose size depends on the number of existing page frames. Each bit of the bitmap of the *k* th entry of either `free_area[0]` or `free_area[1]` describes the status of two buddy blocks of size 2^k page frames. If a bit of the bitmap is equal to 0, either both buddy blocks of the pair are free or both are busy; if it is equal to 1, exactly one of the blocks is busy. When both buddies are free, the kernel treats them as a single free block of size 2^{k+1}.

Let us consider, for sake of illustration, a 128 MB RAM and the bitmaps associated with the non-DMA page frames. The 128 MB can be divided into 32768 single pages, 16384 groups of 2 pages each, or 8192 groups of 4 pages each and so on up to 64 groups of 512 pages each. So the bitmap corresponding to `free_area[0][0]` consists of 16384 bits, one for each pair of the 32768 existing page frames; the bitmap corresponding to `free_area[0][1]` consists of 8192 bits, one for each pair of blocks of two consecutive page frames; the last bitmap corresponding to `free_area[0][9]` consists of 32 bits, one for each pair of blocks of 512 contiguous page frames.

Figure 6-2 illustrates with a simple example the use of the data structures introduced by the buddy system algorithm. The array `mem_map` contains nine free page frames grouped in one block of one (that is, a single page frame) at the top and two blocks of four further down. The double arrows denote doubly linked circular lists implemented by the `next` and `prev` fields. Notice that the bitmaps are not drawn to scale.

Figure 6-2. Data structures used by the buddy system



6.1.2.2 Allocating a block

The `__get_free_ pages( )` function implements the buddy system strategy for allocating page frames. This function checks first whether there are enough free pages, that is, if `nr_free_ pages` is greater than `freepages.min`. If not, it may decide to reclaim page frames (see Section 16.7.4 in Chapter 16). Otherwise, it goes on with the allocation by executing the code included in the `RMQUEUE_TYPE` macro:

```
if (!(gfp_mask & __GFP_DMA))
    RMQUEUE_TYPE(order, 0);
RMQUEUE_TYPE(order, 1);
```

The `order` parameter 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). The second parameter is the index into `free_area`, which is for non-DMA blocks and 1 for DMA blocks. So the code checks `gfp_mask` to see whether non-DMA blocks are allowed and, if so, tries to get blocks from that list (index 0), because it would be better to save DMA blocks for requests that really need them. If the page frames are successfully allocated, the code in the `RMQUEUE_TYPE` macro executes a return statement, thus terminating the _ `_get_free_ pages( )` function. Otherwise, the code in the `RMQUEUE_TYPE` macro is executed again with the second parameter equal to 1, that is, the memory allocation request is satisfied using page frames suitable for DMA.

The code yielded by the `RMQUEUE_TYPE` macro is equivalent to the following fragments. First, a few local variables are declared and initialized:

```
struct free_area_struct * area = &free_area[type][order];
unsigned long new_order = order;
struct page *prev;
struct page *ret;
unsigned long map_nr;
struct page * next;
```

The `type` variable represents the second parameter of the macro: it is equal to when the macro operates on the buddy system for non-DMA page frames and to 1 otherwise.

The macro then 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. This cycle is equivalent to the following structure:

```
do  {
    prev = (struct page *)area;
    ret = prev->next;
    if ((struct page *) area != ret)
        goto block_found;
    new_order++;
    area++;
} while (new_order < 10);
```

If the `while` loop terminates, no suitable free block has been found, so `_ _get_free_pages( )` 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, the corresponding bitmap is updated, and the value of `nr_free_ pages` is decreased:

```
block_found:
    prev->next = ret->next;
    prev->next->prev = prev;
    map_nr = ret-mem_map;
    change_bit(map_nr>>(1+new_order), area->map);
    nr_free_pages -= 1 << order;
    area->count--;
```

If the block found comes from a list of size `new_order` greater than the requested size `order`, a `while` cycle is executed. The rationale behind these lines of codes is the following: 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 last 2^h page frames and iteratively reassigns the first $2^k - 2^h$ page frames to the `free_area` lists having indexes between *h* and *k*.

```
size = 1 << new_order;
while (new_order > order) {
    area--;
    new_order--;
    size >>= 1;
    /* insert *ret as first element in the list
       and update the bitmap */
    next = area->next;
    ret->prev = (struct page *) area;
    ret->next = next;
    next->prev = ret;
    area->next = ret;
    area->count++;
    change_bit(map_nr >> (1+new_order), area->map);
    /* now take care of the second half of
       the free block starting at *ret */
    map_nr += size;
    ret += size;
}
```

Finally, RMQUEUE_TYPE updates the count field of the page descriptor associated with the selected block and executes a return instruction:

```
ret->count = 1;
return PAGE_OFFSET + (map_nr << PAGE_SHIFT);
```

As a result, the __get_free_pages( ) function returns the address of the block found.

6.1.2.3 Freeing a block

The free_ pages_ok( ) function implements the buddy system strategy for freeing page frames. It makes use of three input parameters:

map_nr

The page number of one of the page frames included in the block to be released

order

The logarithmic size of the block

type

Equal to 1 if the page frames are suitable for DMA and to if they are not

The function starts by declaring and initializing a few local variables:

```
struct page * next, * prev;
struct free_area_struct *area = &free_area[type][order];
unsigned long index = map_nr >> (1 + order);
unsigned long mask = (~0UL) << order;
unsigned long flags;
```

The mask variable contains the two's complement of 2^{order}. It is used to transform map_nr into the number of the first page frame of the block to be released and to increment nr_free_ pages:

```
map_nr &= mask;
nr_free_pages -= mask;
```

The function now starts a cycle executed at most (9 - order), 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. The condition driving the while loop is:

```
(mask + (1 << 9))
```

where the single bit set in mask is shifted to the left at each iteration. The body of the loop checks whether the buddy block of the block having number map_nr is free:

```
if (!test_and_change_bit(index, area->map))
    break;
```

If the buddy block is not free, the function breaks out of the cycle; if it is free, the function detaches it from the corresponding list of free blocks. The block number of the buddy is derived from `map_nr` by switching a single bit:

```
area->count--;
next = mem_map[map_nr ^ -mask].next;
prev = mem_map[map_nr ^ -mask].prev;
next->prev = prev;
prev->next = next;
```

At the end of each iteration, the function updates the `mask`, `area`, `index`, and `map_nr` variables:

```
mask <<= 1;
area++;
index >>= 1;
map_nr &= mask;
```

The function then continues the next iteration, trying to merge free blocks twice as large as the ones considered in the previous cycle. When the cycle is finished, the free block obtained cannot be further merged with other free blocks. It is then inserted in the proper list:

```
next = area->next;
mem_map[map_nr].prev = (struct page *) area;
mem_map[map_nr].next = next;
next->prev
= &mem_map[map_nr];
area->next =

&mem_map[map_nr];
area->count++;
```

6.2 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 hundred of bytes?

Clearly, it would be quite wasteful to allocate a full page frame to store a few bytes. The correct 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 Linux 2.0 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%. Following this approach, Linux 2.0 creates 13 geometrically distributed lists of free memory areas whose sizes range from 32 to 131056 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.

6.2.1 The Slab Allocator

Running a memory area allocation algorithm on top of the buddy algorithm is not particularly efficient. Linux 2.2 reexamines the memory area allocation from scratch and comes out with some very clever improvements.

The new algorithm is derived from the *slab allocator* schema developed in 1994 for the Sun Microsystem 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_free_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.

  In order to avoid initializing objects repeatedly, the slab allocator does not discard the objects that have been allocated and then released but saves them in memory. When a new object is then requested, it can be taken from memory without having to be reinitialized.

  In practice, the memory areas handled by Linux do not need to be initialized or deinitialized. For efficiency reasons, Linux does not rely on objects that need constructor or destructor methods; the main motivation for introducing a slab allocator is to reduce the number of calls to the buddy system allocator. Thus, although the kernel fully supports the constructor and destructor methods, the pointers to these methods are NULL.

- 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. Since processes are created and destroyed quite frequently, previous versions of the Linux kernel wasted time allocating and deallocating the page frames containing the same memory areas repeatedly; in Linux 2.2 they are saved in a cache and reused instead.
- 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 having 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 Linux 2.0), even if this approach leads to internal fragmentation.

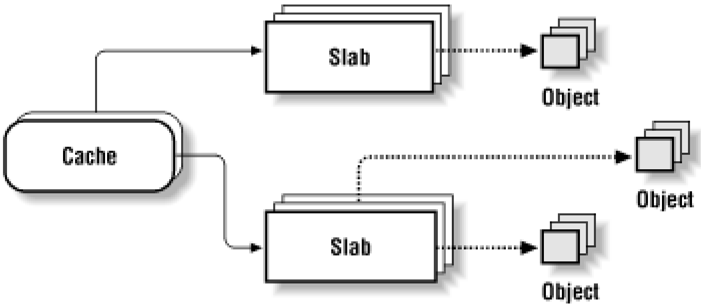
- There is another subtle bonus in introducing objects whose sizes are not geometrically distributed: the initial addresses of the data structures are less prone to be concentrated on physical addresses whose values are a power of 2. This, in turn, leads to better performance by the processor hardware 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.[1]

[1] The impact of a kernel function on the hardware cache is denoted as 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, since 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 slab allocator caches used by Linux may be viewed at runtime by reading the */proc/slabinfo* file.

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 6-3).

Figure 6-3. The slab allocator components



The slab allocator never releases the page frames of an empty slab on its own. It would not know when free memory is needed, and there is no benefit to releasing objects when there is still plenty of free memory for new objects. Therefore, releases occur only when the kernel is looking for additional free page frames (see tSection 6.2.12 later in this chapter and Section 16.7 in Chapter 16).

6.2.2 Cache Descriptor

Each cache is described by a table of type `struct kmem_cache_s` (which is equivalent to the type `kmem_cache_t`). The most significant fields of this table are:

`c_name`

Points to the name of the cache.

`c_firstp`, `c_lastp`

Point, respectively, to the first and last slab descriptor of the cache. The slab descriptors of a cache are linked together through a doubly linked, circular, partially ordered list: the first elements of the list include slabs with no free objects, then come the slabs that include used objects along with at least one free object, and finally the slabs that include only free objects.

`c_freep`

Points to the `s_nextp` field of the first slab descriptor that includes at least one free object.

`c_num`

Number of objects packed into a single slab. (All slabs of the cache have the same size.)

`c_offset`

Size of the objects included in the cache. (This size may be rounded up if the initial addresses of the objects must be memory aligned.)

`c_ gfporder`

Logarithm of the number of contiguous page frames included in a single slab.

`c_ctor , c_dtor`

Point, respectively, to the constructor and destructor methods associated with the cache objects. They are currently set to `NULL`, as stated earlier.

`c_nextp`

Points to the next cache descriptor. All cache descriptors are linked together in a simple list by means of this field.

`c_flags`

An array of flags that describes some permanent properties of the cache. There is, for instance, a flag that specifies which of two possible alternatives (see the following section) has been chosen to store the object descriptors in memory.

`c_magic`

A magic number selected from a predefined set of values. Used to check both the current state of the cache and its consistency.

6.2.3 Slab Descriptor

Each slab of a cache has its own descriptor of type `struct kmem_slab_s` (equivalent to the type `kem_slab_t`).

Slab descriptors can be stored in two possible places, the choice depending normally on the size of the objects in the slab. If the object size is smaller than 512 bytes, the slab descriptor is stored at the end of the slab; otherwise, it is stored outside of the slab. The latter option is preferable for large objects whose sizes are a submultiple of the slab size. In some cases, the kernel may violate this rule by setting the `c_flags` field of the cache descriptor differently.

The most significant fields of a slab descriptor are:

`s_inuse`

Number of objects in the slab that are currently allocated.

`s_mem`

Points to the first object (either allocated or free) inside the slab.

`s_freep`

Points to the first free object (if any) in the slab.

`s_nextp`, `s_prevp`

Point, respectively, to the next and previous slab descriptor. The `s_nextp` field of the last slab descriptor in the list points to the `c_offset` field of the corresponding cache descriptor.

`s_dma`

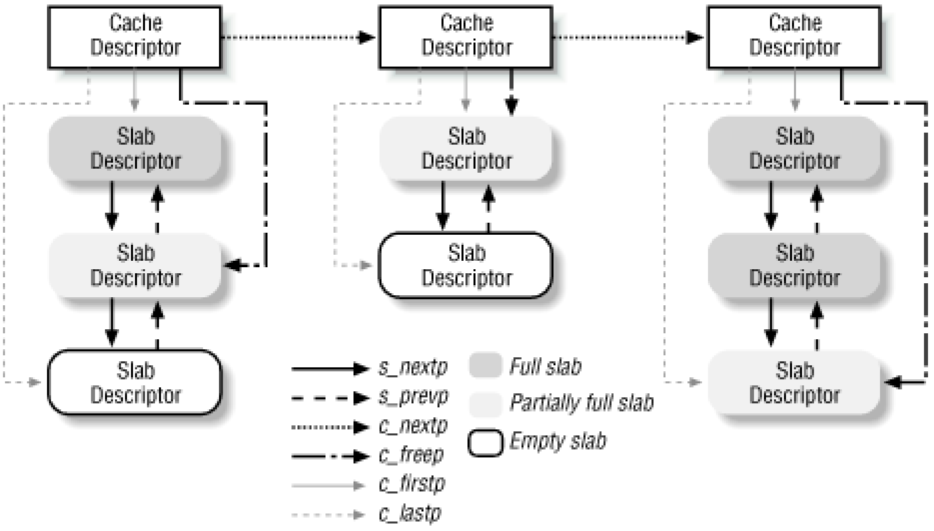
Flag set if the objects included in the slab can be used by the DMA processor.

`s_magic`

Similar to the `c_magic` field of the cache descriptor. It contains a magic number selected from a predefined set of values and is used to check both the current state of the slab and its consistency. The values of this field are different from those of the corresponding `c_magic` field of the cache descriptor. The offset of `s_magic` within the slab descriptor is equal to the offset of `c_magic` with respect to `c_offset` inside the cache descriptor; the checking routine relies on their being the same.

Figure 6-4 illustrates the major relationships between cache and slab descriptors. Full slabs precede partially full slabs that precede empty slabs.

Figure 6-4. Relationships between cache and slab descriptors



6.2.4 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.

The general caches are:

- A first cache contains the cache descriptors of the remaining caches used by the kernel. The `cache_cache` variable contains its descriptor.
- A second cache contains the slab descriptors that are not stored inside the slabs. The `cache_slabp` variable points to its descriptor.
- Thirteen additional caches contain geometrically distributed memory areas. The table called `cache_sizes` whose elements are of type `cache_sizes_t` points to the 13 cache descriptors associated with memory areas of size 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and 131072 bytes, respectively. The table `cache_sizes` is used to efficiently derive the cache address corresponding to a given size.

The `kmem_cache_init( )` and `kmem_cache_sizes_init( )` functions are 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 creates a new cache descriptor for the new cache and inserts the descriptor in the `cache_cache` general cache. It should be noted that once a cache has been created, it cannot be destroyed.

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.

6.2.5 Interfacing the Slab Allocator with the Buddy System

When the slab allocator creates new slabs, it relies on the buddy system algorithm to obtain a group of free contiguous page frames. To that purpose, it invokes the kmem_getpages( ) function:

```
void * kmem_getpages(kmem_cache_t *cachep,
                     unsigned long flags, unsigned int *dma)
{
    void    *addr;
    *dma = flags & SLAB_DMA;
    addr = (void*) __get_free_pages(flags, cachep->c_gfporder);
    if (!*dma && addr) {
        struct page *page = mem_map + MAP_NR(addr);
        *dma = 1<<cachep->c_gfporder;
        while ((*dma)--) {
            if (!PageDMA(page)) {
                *dma = 0;
                break;
            }
            page++;
        }
    }
    return addr;
}
```

The 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 in the cachep->c_gfporder field)

flags

Specifies how the page frame is requested (see Section 6.1.1 earlier in this chapter)

dma

Points to a variable that is set to 1 by kmem_getpages( ) if the allocated page frames are suitable for ISA DMA

In the reverse operation, page frames assigned to a slab allocator can be released (see Section 6.2.7 later in this chapter) by invoking the kmem_freepages( ) function:

```
void kmem_freepages(kmem_cache_t *cachep, void *addr)
{
    unsigned long i = (1<<cachep->c_gfporder);
    struct page *page = &mem_map[MAP_NR(addr)];
    while (i--) {
        PageClearSlab(page);
        page++;
    }
    free_pages((unsigned long)addr, cachep->c_gfporder);
}
```

The function releases the page frames, starting from the one having physical address addr, that had been allocated to the slab of the cache identified by cachep.

6.2.6 Allocating a Slab to a Cache

A newly created cache does not contain any slab and therefore no 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 any free object.

When this occurs, the slab allocator assigns a new slab to the cache by invoking kmem_cache_grow( ). This function calls kmem_ getpages( ) to obtain a group of page frames from the buddy system; it then calls kmem_cache_slabmgmt( ) to get a new slab descriptor. Next, it calls kmem_cache_init_objs( ), which applies the constructor method (if defined) to all the objects contained in the new slab. It then calls kmem_slab_link_end( ), which inserts the slab descriptor at the end of the cache slab list:

```
void kmem_slab_link_end(kmem_cache_t *cachep,
                        kmem_slab_t *slabp)
{
    kmem_slab_t *lastp = cachep->c_lastp;
    slabp->s_nextp = kmem_slab_end(cachep);
    slabp->s_prevp = lastp;
    cachep->c_lastp = slabp;
    lastp->s_nextp = slabp;
}
```

The kmem_slab_end macro yields the address of the c_offset field of the corresponding cache descriptor (as stated before, the last element of a slab list points to that field).

After inserting the new slab descriptor into the list, kmem_cache_ grow( ) loads the next and prev fields, respectively, of the descriptors of all page frames included in the new slab with the address of the cache descriptor and the address of the slab descriptor. This works correctly because the next and prev fields are used by functions of the buddy system only when the page frame is free, while page frames handled by the slab allocator functions are not free as far as the buddy system is concerned. Therefore, the buddy system will not be confused by this specialized use of the page frame descriptor.

6.2.7 Releasing a Slab from a Cache

As stated previously, the slab allocator never releases the page frames of an empty slab on its own. In fact, a slab is released only if both the following conditions hold:

- The buddy system is unable to satisfy a new request for a group of page frames.
- The slab is empty, that is, all the objects included in it are free.

When the kernel looks for additional free page frames, it calls try_to_free_pages( ); this function, in turn, may invoke kmem_cache_reap( ), which selects a cache that contains at least one empty slab. The kmem_slab_unlink( ) function then removes the slab from the cache list of slabs:

```
void kmem_slab_unlink(kmem_slab_t *slabp)
{
    kmem_slab_t *prevp = slabp->s_prevp;
    kmem_slab_t *nextp = slabp->s_nextp;
    prevp->s_nextp = nextp;
    nextp->s_prevp = prevp;
}
```

Subsequently, the slab—together with the objects in it—is destroyed by invoking `kmem_slab_destroy( )`:

```
void kmem_slab_destroy(kmem_cache_t *cachep, kmem_slab_t *slabp)
{
    if (cachep->c_dtor) {
        unsigned long num = cachep->c_num;
        void *objp = slabp->s_mem;
        do {
            (cachep->c_dtor)(objp, cachep, 0);
            objp += cachep->c_offset;
            if (!slabp->s_index)
                objp += sizeof(kmem_bufctl_t);
        } while (--num);
    }
    slabp->s_magic = SLAB_MAGIC_DESTROYED;
    if (slabp->s_index)
        kmem_cache_free(cachep->c_index_cachep, slabp->s_index);
    kmem_freepages(cachep, slabp->s_mem-slabp->s_offset);
    if (SLAB_OFF_SLAB(cachep->c_flags))
        kmem_cache_free(cache_slabp, slabp);
}
```

The function checks whether the cache has a destructor method for its objects (the `c_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 (in this case the `s_index` and `c_index_cachep` fields are not NULL, as explained later in this chapter), the function releases it from the cache of the slab descriptors.

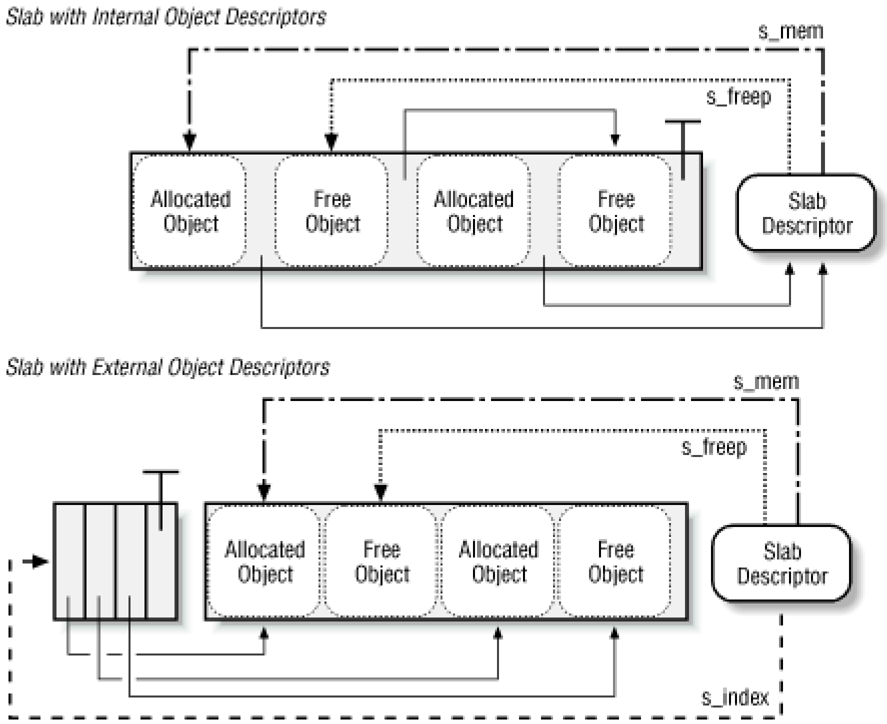
Some modules of Linux (see Appendix B) may create caches. In order to avoid wasting memory space, the kernel must destroy all slabs in all caches created by a module before removing it.[2] The `kmem_cache_shrink( )` function destroys all the slabs in a cache by invoking `kmem_slab_destroy( )` iteratively. The `c_ growing` field of the cache descriptor is used to prevent `kmem_cache_shrink( )` from shrinking a cache while another kernel control path attempts to allocate a new slab for it.

[2] We stated previously that Linux does not destroy caches. Thus, when linking in a new module, the kernel must check whether the new cache descriptors requested by it were already created in a previous installation of that module or another one.

6.2.8 Object Descriptor

Each object has a descriptor of type `struct kmem_bufctl_s` (equivalent to the type `kmem_bufctl_t`). Like the slab descriptors themselves, the object descriptors of a slab can be stored in two possible ways, illustrated by Figure 6-5.

Figure 6-5. Relationships between slab and object descriptors



External object descriptors

Stored outside the slab, in one of the general caches pointed to by `cache_sizes`. In this case, the first object descriptor in the memory area describes the first object in the slab and so on. 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 (`c_num` field of the cache descriptor). The cache containing the objects themselves is tied to the cache containing their descriptors through two fields. First, the `c_index_cachep` field of the cache containing the slab points to the cache descriptor of the cache containing the object descriptors. Second, the `s_index` field of the slab descriptor points to the memory area containing the object descriptors.

Internal object descriptors

Stored inside the slab, right after the objects they describe. In this case, the `c_index_cachep` field of the cache descriptor and the `s_index` field of the slab descriptor are both NULL.

The slab allocator chooses the first solution when the size of the objects is a multiple of 512, 1024, 2048, or 4096: in this case, storing control structures inside the slab would result in a high level of internal fragmentation. If the size of the objects is smaller than 512 bytes or not a multiple of 512, 1024, 2048, or 4096 the slab allocator stores the object descriptors inside the slab.

Object descriptors are simple structures consisting of a single field:

```
typedef struct kmem_bufctl_s {
    union {
        struct kmem_bufctl_s * buf_nextp;
        kmem_slab_t *          buf_slabp;
        void *                 buf_objp;
    } u;
} kmem_bufctl_t;
#define    buf_nextp   u.buf_nextp
#define    buf_slabp   u.buf_slabp
#define    buf_objp    u.buf_objp
```

This field has the following meaning, depending on the state of the object and the locations of the object descriptors:

`buf_nextp`

If the object is free, it points to the next free object in the slab, thus implementing a simple list of free objects inside the slab.

`buf_objp`

If the object is allocated and its object descriptor is stored outside of the slab, it points to the object.

`buf_slabp`

If the object is allocated and its object descriptor is stored inside the slab, it points to the slab descriptor of the slab in which the object is stored. This holds whether the slab descriptor is stored inside or outside of the slab.

Figure 6-5 illustrates the relationships among slabs, slab descriptors, objects, and object descriptors. Notice that, although the figure suggests that the slab descriptor is stored outside of the slab, it remains unchanged if the descriptor is stored inside it.

6.2.9 Aligning Objects in Memory

The objects managed by the slab allocator can be *aligned* in memory, that is, they can be stored in memory cells whose initial physical addresses are multiples of a given constant, usually a power of 2. This constant is called the *alignment factor*, and its value is stored in the `c_align` field of the cache descriptor. The `c_offset` field, which contains the object size, takes into account the number of padding bytes added to obtain the proper alignment. If the value of `c_align` is 0, no alignment is required for the objects.

The largest alignment factor allowed by the slab allocator is 4096, that is, the page frame size. This means that objects can be aligned by referring either to their physical addresses or to 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, the `kmem_cache_create( )` function attempts to align objects according to the word size specified by the `BYTES_PER_WORD` macro. For Intel Pentium processors, the macro yields the value 4 because the word is 32 bits long. However, the function does not align objects if this leads to a consistent waste of memory.

When creating a new cache, it's possible to specify that the objects included in it be aligned in the first-level cache. To achieve this, set 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 factor of `L1_CACHE_BYTES`; this ensures that an 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.

6.2.10 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 tend to be 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 us consider a cache whose objects are aligned in RAM. Thus, the `c_align` field of the cache descriptor has a positive value, say *aln*. Even taking into account the alignment constraint, 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 `c_num` field of the cache descriptor).

osize

Object size including the alignment bytes (its value is in the `c_offset` field) plus object descriptor size (if the descriptor is contained inside the slab).

dsize

Slab descriptor size; its value is equal to if the slab descriptor is 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:

$$slab\ length = (num \times osize)+dsize+free$$

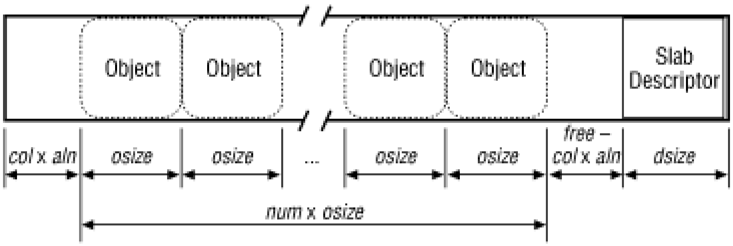
free is always smaller than *osize*, since otherwise it would be possible to place additional objects inside the slab. However, *free* could be greater than *aln*.

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*+1. The first color is denoted as and the last one (whose value is in the `c_colour` field of the cache descriptor) is denoted as *free/aln*.

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* x *aln* bytes; this value is stored in the `s_offset` field of the slab descriptor. Figure 6-6 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 6-6. 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 having the color 0, that is, 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 `c_colour_next`. The `kmem_cache_grow( )` function assigns the color specified by `c_colour_next` to a new slab and then decrements the value of this field. After reaching 0, it wraps around again to the maximum available value:

```
if (!(offset = cachep->c_colour_next--))
    cachep->c_colour_next = cachep->c_colour;
offset *= cachep->c_align;
slabp->s_offset = offset;
```

In this way, each slab is created with a different color from the previous one, up to the maximum available colors.

6.2.11 Allocating an Object to a Cache

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. `kmem_cache_alloc( )` first checks whether the cache descriptor exists; it then retrieves from the `c_freep` field the address of the `s_nextp` field of the first slab that includes at least one free object:

```
slabp = cachep->c_freep;
```

If `slabp` does not point to a slab, it then jumps to `alloc_new_slab` and invokes `kmem_cache_grow( )` to add a new slab to the cache:

```
if (slabp->s_magic != SLAB_MAGIC_ALLOC)
    goto alloc_new_slab;
```

The value `SLAB_MAGIC_ALLOC` in the `s_magic` field indicates that the slab contains at least one free object. If the slab is full, `slabp` points to the `cachep->c_offset` field, and thus `slabp->s_magic` coincides with `cachep->c_magic`: in this case, however, this field contains a magic number for the cache different from `SLAB_MAGIC_ALLOC`.

After obtaining a slab with a free object, the function increments the counter containing the number of objects currently allocated in the slab:

```
slabp->s_inuse++;
```

It then loads `bufp` with the address of the first free object inside the slab and, correspondingly, updates the `slabp->s_freep` field of the slab descriptor to point to the next free object:

```
bufp = slabp->s_freep;
slabp->s_freep = bufp->buf_nextp;
```

If `slabp->s_freep` becomes NULL, the slab no longer includes free objects, so the `c_freep` field of the cache descriptor must be updated:

```
if (!slabp->s_freep)
    cachep->c_freep = slabp->s_nextp;
```

Notice that there is no need to change the position of the slab descriptor inside the list since it remains partially ordered. Now the function must derive the address of the free object and update the object descriptor.

If the `slabp->s_index` field is null, the object descriptors are stored right after the objects inside the slab. In this case, the address of the slab descriptor is first stored in the object descriptor's single field to denote the fact that the object is no longer free; then the object address is derived by subtracting from the address of the object descriptor the object size included in the `cachep->c_offset` field:

```
if (!slabp->s_index) {
        bufp->buf_slabp = slabp;
        objp = ((void*)bufp) - cachep->c_offset;
    }
```

If the `slabp->s_index` field is not zero, it points to a memory area outside of the slab where the object descriptors are stored. In this case, the function first computes the relative position of the object descriptor in the outside memory area; it then multiplies this number by the object size; finally, it adds the result to the address of the first object in the slab, thus yielding the address of the object to be returned. As in the previous case, the object descriptor single field is updated and points now to the object:

```
if (slabp->s_index) {
    objp = ((bufp-slabp->s_index)*cachep->c_offset) +
                slabp->s_mem;
    bufp->buf_objp = objp;
}
```

The function terminates by returning the address of the new object:

```
return objp;
```

6.2.12 Releasing an Object from a Cache

The `kmem_cache_free( )` function releases an object previously obtained by the slab allocator. Its parameters are `cachep`, the address of the cache descriptor, and `objp`, the address of the object to be released. The function starts by checking the parameters, after which it determines the address of the object descriptor and that of the slab containing the object. It uses the `cachep->c_flags` flag, included in the cache descriptor, to determine whether the object descriptor is located inside or outside of the slab.

In the former case, it determines the address of the object descriptor by adding the object's size to its initial address. The address of the slab descriptor is then extracted from the appropriate field in the object descriptor:

```
if (!SLAB_BUFCTL(cachep->c_flags)) {
    bufp = (kmem_bufctl_t *)(objp+cachep->c_offset);
    slabp = bufp->buf_slabp;
}
```

In the latter case, it determines the address of the slab descriptor from the `prev` field of the descriptor of the page frame containing the object (refer to Section 6.2.6 for the role of `prev`). The address of the object descriptor is derived by first computing the sequence number of the object inside the slab (object address minus first object address divided by object length). This number is then used to determine the position of the object descriptor starting from the beginning of the outside area pointed to by the `slabp->s_index` field of the slab descriptor. To be on the safe side, the function checks that the object's address passed as a parameter coincides with the address that its object descriptor says it should have:

```
if (SLAB_BUFCTL(cachep->c_flags)) {
    slabp = (kmem_slab_t *)((&mem_map[MAP_NR(objp)])->prev);
    bufp =    &slabp->s_index[(objp - slabp->s_mem) /
                                  cachep->c_offset];
    if (objp != bufp->buf_objp)
        goto bad_obj_addr;
}
```

Now the function checks whether the `slabp->s_magic` field of the slab descriptor contains the correct magic number and whether the `slabp->s_inuse` field is greater than 0. If everything is okay, it decrements the value of `slabp->s_inuse` and inserts the object into the slab list of free objects:

```
slabp->s_inuse--;
bufp->buf_nextp = slabp->s_freep;
slabp->s_freep = bufp;
```

If `bufp->buf_nextp` is NULL, the list of free objects includes only one element: the object that is being released. In this case, the slab was previously filled to capacity and it might be necessary to reinsert its slab descriptor in a new position in the list of slab descriptors. (Remember that completely filled slabs appear before slabs with some free objects in the partially ordered list.) This is done by the `kmem_cache_one_free( )` function:

```
if (!bufp->buf_nextp)
    kmem_cache_one_free(cachep, slabp);
```

If the slab includes other free objects besides the one being released, it is necessary to check whether *all* objects are free. As in the previous case, this would make it necessary to reinsert the slab descriptor in a new position in the list of slab descriptors. The move is done by the `kmem_cache_full_free( )` function:

```
if (bufp->buf_nextp)
    if (!slabp->s_inuse)
        kmem_cache_full_free(cachep, slabp);
```

The `kmem_cache_free( )` function terminates here.

6.2.13 General Purpose Objects

As stated in Section 6.1.2, 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 131072 bytes.

Objects of this type are obtained by invoking the `kmalloc( )` function:

```
void * kmalloc(size_t size, int flags)
{
    cache_sizes_t *csizep = cache_sizes;
    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        return __kmem_cache_alloc(csizep->cs_cachep, flags);
    }
    printk(KERN_ERR "kmalloc: Size (%lu) too large\n",
                    (unsigned long) size);
    return NULL;
}
```

The function uses the `cache_sizes` table to locate the cache descriptor of the cache containing objects of the right size. It then calls `kmem_cache_alloc( )` to allocate the object.[3]

[3] Actually, for efficiency reasons, the code of `kmem_cache_alloc()` is copied inside the body of `kmalloc()`. The `__kmem_cache_alloc()` function, which implements `kmem_cache_alloc()`, is declared `inline`.

Objects obtained by invoking `kmalloc( )` can be released by calling `kfree( )`:[4]

[4] A similar function called `kfree_s( )` requires an additional parameter, namely, the size of the object to be released. This function was used in previous versions of Linux where the size of the memory area had to be determined before releasing it. It is still used by some modules of the filesystem.

```
void kfree(const void *objp)
{
    struct page *page;
    int    nr;
    if (!objp)
        goto null_ptr;
    nr = MAP_NR(objp);
    if (nr >= num_physpages)
        goto bad_ptr;
    page = &mem_map[nr];
    if (PageSlab(page)) {
        kmem_cache_t    *cachep;
        cachep = (kmem_cache_t *)(page->next);
        if (cachep && (cachep->c_flags & SLAB_CFLGS_GENERAL)) {
            __kmem_cache_free(cachep, objp);
            return;
        }
    }
bad_ptr:
    printk(KERN_ERR "kfree: Bad obj %p\n", objp);
    *(int *) 0 = 0; /* FORCE A KERNEL DUMP */
null_ptr:
    return;
}
```

The proper cache descriptor is identified by reading the `next` field of the descriptor of the first page frame containing the memory area. If this field points to a valid descriptor, the memory area is released by invoking `kmem_cache_free( )`.

6.3 Noncontiguous Memory Area Management

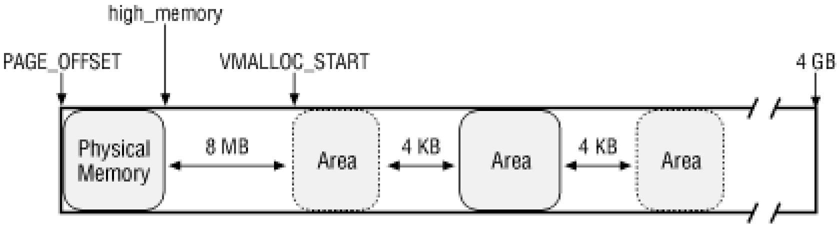
We already know from an earlier discussion that it is preferable to map memory areas into sets of contiguous page frames, thus making better use of the cache and achieving lower average memory access times. Nevertheless, if the requests for memory areas are infrequent, it makes sense to consider an allocation schema 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 4096. Linux uses noncontiguous memory areas sparingly, for instance, to allocate data structures for active swap areas (see Section 16.2.3 in Chapter 16), to allocate space for a module (see Appendix B), or to allocate buffers to some I/O drivers.

6.3.1 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). We learned in the Chapter 2 in Section 2.5.4 that the kernel reserved this whole upper area of memory to map available RAM for kernel use. But available RAM occupies only a small fraction of the gigabyte, starting at the `PAGE_OFFSET` address. All the linear addresses above that reserved area are available for mapping noncontiguous memory areas. The linear address that corresponds to the end of physical memory is stored in the `high_memory` variable.

Figure 6-7 shows how linear addresses are assigned to noncontiguous memory areas. A safety interval of size 8 MB (macro `VMALLOC_OFFSET`) is inserted between the end of the physical memory 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 6-7. 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. It is defined as follows:

```
#define VMALLOC_START (((unsigned long) high_memory + \
                        VMALLOC_OFFSET) & ~(VMALLOC_OFFSET-1))
```

6.3.2 Descriptors of Noncontiguous Memory Areas

Each noncontiguous memory area is associated with a descriptor of type `struct vm_struct`:

```
struct vm_struct {
    unsigned long flags;
    void * addr;
    unsigned long size;
    struct vm_struct * next;
};
```

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. The `addr` field contains the linear address of the first memory cell of the area; the `size` field contains the size of the area plus 4096 (the size of the previously mentioned interarea safety interval).

The `get_vm_area( )` function creates new descriptors of type `struct vm_struct`; its parameter `size` specifies the size of the new memory area:

```
struct vm_struct * get_vm_area(unsigned long size)
{
    unsigned long addr;
    struct vm_struct **p, *tmp, *area;
    area = (struct vm_struct *) kmalloc(sizeof(*area),
                                        GFP_KERNEL);
    if (!area)
        return NULL;
    addr = VMALLOC_START;
    for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
        if (size + addr < (unsigned long) tmp->addr)
            break;
        addr = tmp->size + (unsigned long) tmp->addr;
        if (addr > 0xffffd000-size) {
            kfree(area);
            return NULL;
        }
    }
    area->addr = (void *)addr;
    area->size = size + PAGE_SIZE;
    area->next = *p;
    *p = area;
    return area;
}
```

The function first calls `kmalloc( )` to obtain a memory area for the new descriptor. It then scans the list of descriptors of type `struct vm_struct` looking for an available range of linear addresses that includes at least `size+4096` addresses. If such an interval exists, the function initializes the fields of the descriptor and terminates by returning the initial address of the noncontiguous memory area. Otherwise, when `addr + size` exceeds the 4 GB limit, `get_vm_area( )` releases the descriptor and returns NULL.

6.3.3 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, then it returns the initial linear address of the new area; otherwise, it returns a NULL pointer:

```
void * vmalloc(unsigned long size)
{
    void * addr;
    struct vm_struct *area;
    size = (size+PAGE_SIZE-1)&PAGE_MASK;
    if (!size || size > (num_physpages << PAGE_SHIFT))
        return NULL;
    area = get_vm_area(size);
    if (!area)
        return NULL;
    addr = area->addr;
    if (vmalloc_area_pages((unsigned long) addr, size)) {
        vfree(addr);
        return NULL;
    }
    return addr;
}
```

The function starts by rounding up the value of the `size` parameter to a multiple of 4096 (the page frame size). It also performs a sanity check to make sure the size is greater than and less than or equal to the existing number of page frames. If the size fits available memory, `vmalloc( )` invokes `get_vm_area( )`, which creates a new descriptor and returns the linear addresses assigned to the memory area. Then `vmalloc( )` invokes `vmalloc_area_pages( )` to request noncontiguous page frames and terminates by returning the initial linear address of the noncontiguous memory area.

The `vmalloc_area_pages( )` function makes use of two parameters: `address`, the initial linear address of the area, and `size`, its size. The linear address of the end of the area is assigned to the `end` local variable:

```
end = address + size;
```

The function then uses the `pgd_offset_k` macro to derive the entry in the Page Global Directory related to the initial linear address of the area:

```
dir = pgd_offset_k(address);
```

The function then executes the following cycle:

```
while (address < end) {
    pmd_t *pmd = pmd_alloc_kernel(dir, address);
    if (!pmd)
        return -ENOMEM;
    if (alloc_area_pmd(pmd, address, end - address))
        return -ENOMEM;
    set_pgdir(address, *dir);
    address = (address + PGDIR_SIZE) & PGDIR_MASK;
    dir++;
}
```

In each cycle, it first invokes `pmd_alloc_kernel( )` to create a Page Middle Directory for the new area. It then calls `alloc_area_pmd( )` to allocate all the Page Tables associated with the new Page Middle Directory. Next, it invokes `set_pgdir( )` to update the entry corresponding to the new Page Middle Directory in all existing Page Global Directories (see Section 2.5.4 in Chapter 2). It adds the constant 2^{22}, that is, the size of the range of linear addresses spanned by a single Page Middle Directory, to the current value of `address`, and it increases the pointer `dir` to the Page Global Directory.

The cycle is repeated until all page table entries referring to the noncontiguous memory area have been set up.

The `alloc_area_pmd( )` function executes a similar cycle for all the Page Tables that a Page Middle Directory points to:

```
while (address < end) {
    pte_t * pte = pte_alloc_kernel(pmd, address);
    if (!pte)
        return -ENOMEM;
    if (alloc_area_pte(pte, address, end - address))
        return -ENOMEM;
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
}
```

The `pte_alloc_kernel( )` function (see Section 2.5.2 in Chapter 2) allocates a new Page Table and updates the corresponding entry in the Page Middle Directory. Next, `alloc_area_pte( )` allocates all the page frames corresponding to the entries in the Page Table. The value of `address` is increased by 2^{22}, that is, the size of the linear address interval spanned by a single Page Table, and the cycle is repeated.

The main cycle of `alloc_area_pte( )` is:

```
while (address < end) {
    unsigned long page;
    if (!pte_none(*pte))
        printk("alloc_area_pte: page already exists\n");
    page = __ get_free_page(GFP_KERNEL);
    if (!page)
        return -ENOMEM;
    set_pte(pte, mk_pte(page, PAGE_KERNEL));
    address += PAGE_SIZE;
    pte++;
}
```

Each page frame is allocated through `__get_free_page( )`. 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 4096, that is, the length of a page frame, to `address`.

6.3.4 Releasing a Noncontiguous Memory Area

The `vfree( )` function releases noncontiguous memory areas. Its parameter `addr` contains the initial linear address of the area to be released. `vfree( )` first scans the list pointed by `vmlist` to find the address of the area descriptor associated with the area to be released:

```
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        *p = tmp->next;
        vmfree_area_pages((unsigned long)(tmp->addr),
                          tmp->size);
        kfree(tmp);
        return;
    }
}
printk("Trying to vfree(  ) nonexistent vm area (%p)\n", addr);
```

The `size` field of the descriptor specifies the size of the area to be released. The area itself is released by invoking `vmfree_area_pages( )`, while the descriptor is released by invoking `kfree( )`.

The `vmfree_area_pages( )` function takes two parameters: the initial linear address and the size of the area. It executes the following cycle to reverse the actions performed by `vmalloc_area_pages( )`:

```
while (address < end) {
    free_area_pmd(dir, address, end - address);
    address = (address + PGDIR_SIZE) & PGDIR_MASK;
    dir++;
}
```

In turn, `free_area_pmd( )` reverses the actions of `alloc_area_pmd( )` in the cycle:

```
while (address < end) {
    free_area_pte(pmd, address, end - address);
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
}
```

Again, `free_area_pte( )` reverses the activity of `alloc_area_pte( )` in the cycle:

```
while (address < end) {
    pte_t page = *pte;
    pte_clear(pte);
    address += PAGE_SIZE;
    pte++;
    if (pte_none(page))
        continue;
    if (pte_present(page)) {
        free_page(pte_page(page));
        continue;
    }
    printk("Whee... Swapped out page in kernel page table\n");
}
```

Each page frame assigned to the noncontiguous memory area is released by means of the buddy system `free_ page( )` function. The corresponding entry in the Page Table is set to by the `pte_clear` macro.

6.4 Anticipating Linux 2.4

Linux 2.2 has two buddy systems: the first one handles page frames suitable for ISA DMA, while the second one handles page frames not suitable for ISA DMA. Linux 2.4 adds a third buddy system for the high physical memory, that is, for the page frames not permanently mapped by the kernel. Using a high-memory page frame implies changing an entry in a special kernel Page Table to map the page frame physical addresses in the 4 GB linear address space.

Actually, Linux 2.4 views the three portions of RAM as different "zones." Each zone has its own counters and watermarks to monitor the number of free page frames. When a memory allocation request takes place, the kernel first tries to fetch the page frames from the most suitable zone; if it fails, it may fall back on another zone.

The slab allocator is mostly unchanged. However, Linux 2.4 allows a slab allocator cache that is no longer useful to be destroyed. Recall that in Linux 2.2 a slab allocator cache can be dynamically created but not destroyed. Modules that create their own slab allocator cache when loaded are now expected to destroy it when unloaded.

Chapter 8. System Calls

Operating systems offer processes running in User Mode a set of interfaces to interact with hardware devices such as the CPU, disks, printers, and so on. Putting an extra layer between the application and the hardware has several advantages. First, it makes programming easier, freeing users from studying low-level programming characteristics of hardware devices. Second, it greatly increases system security, since the kernel can check the correctness of the request at the interface level before attempting to satisfy it. Last but not least, these interfaces make programs more portable since they can be compiled and executed correctly on any 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 system calls are implemented by the Linux kernel.

8.1 POSIX APIs and System Calls

Let us 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 programmers. Some of the APIs defined by the *libc* standard C library refer to *wrapper routines*, that is, routines whose only purpose is to issue a system call. Usually, each system call corresponds to a wrapper routine; the wrapper routine defines the API that application programs should refer to.

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 like math functions, there may be no reason to make system 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( )` POSIX APIs are implemented in the *libc* library: the code in that library keeps track of the allocation and deallocation requests and uses the `brk( )` system call in order to enlarge or shrink the process heap (see Section 7.6 in Chapter 7).

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 since 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 since 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 denotes in most cases, but not always, 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 associated with a macro, which yields a corresponding positive integer value. The POSIX standard specifies the macro names of several error codes. In Linux on Intel 80x86 systems, those macros are defined in a header file called *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.

8.2 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. In Linux the system calls must be invoked by executing the `int $0x80` Assembly instruction, which raises the programmed exception having vector 128 (see Section 4.4.1 and Section 4.2.5 in Chapter 4).

Since the kernel implements many different system calls, the process must pass a parameter called the *system call number* to identify the required system call; the `eax` register is used for that purpose. As we shall see in Section 8.2.3 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 null 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. The `errno` variable is not set or used by the kernel.

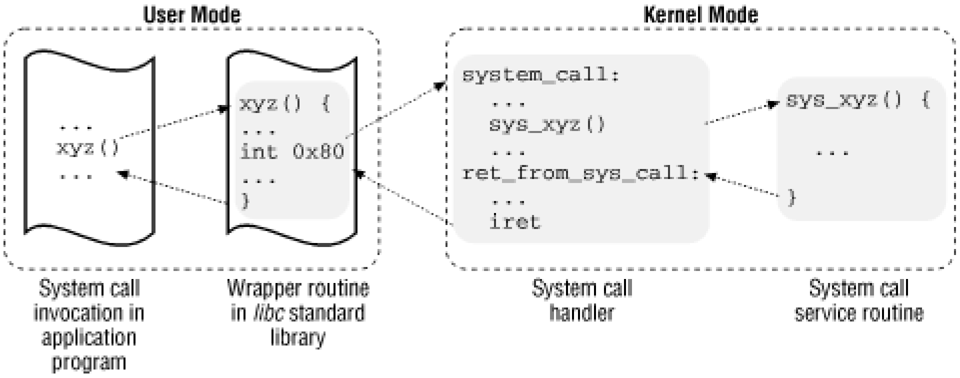
The system call handler, which has a structure similar to that of the other exception handlers, performs the following operations:

- 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 by means of the `ret_from_sys_call( )` function (this function 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 8-1 illustrates the relationships among 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.

Figure 8-1. Invoking a system call



In order to associate each system call number with its corresponding service routine, the kernel makes use of a *system call dispatch table* ; this table is stored in the `sys_call_table` array and has `NR_syscalls` entries (usually 256): the *n*th 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, any 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`.

8.2.1 Initializing System Calls

The `trap_init( )` function, invoked during kernel initialization, sets up the IDT 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 Section 4.4.1 in Chapter 4):

Segment Selector

The `__KERNEL_CS` Segment Selector of the kernel code segment.

Offset

Pointer to the `system_call( )` exception 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 Section 4.2.5 in Chapter 4).

8.2.2 The system_call( ) Function

The `system_call( )` function implements the system call handler. It 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 Section 4.2.5 in Chapter 4). The `SAVE_ALL` macro, which was already discussed in Section 4.6.3 in Chapter 4, also loads the Segment Selector of the kernel data segment in `ds` and `es`:

```
system_call:
  pushl %eax
  SAVE_ALL
  movl %esp, %ebx
  andl $0xffffe000, %ebx
```

The function also stores in `ebx` the address of the `current` process descriptor; this is done by taking the value of the kernel stack pointer and rounding it up to a multiple of 8 KB (see Section 3.1.2 in Chapter 3).

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 `NR_syscalls`, the system call handler terminates:

```
cmpl $(NR_syscalls), %eax
  jb nobadsys
  movl $(-ENOSYS), 24(%esp)
  jmp ret_from_sys_call
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 (at offset 24 from the current stack top). It then jumps to `ret_from_sys_call( )`. In this way, when the process resumes its execution in User Mode, it will find a negative return code in `eax`.

Next, the `system_call( )` function checks whether the `PF_TRACESYS` flag included in the `flags` field of `current` is equal to 1, that is, whether the system call invocations of the executed program are being traced by some debugger. If this is the case, `system_call( )` invokes the `syscall_trace( )` function twice, once right before and once right after the execution of the system call service routine. This function stops `current` and thus allows the debugging process to collect information about it.

Finally, the specific service routine associated with the system call number contained in `eax` is invoked:

```
call *sys_call_table(0, %eax, 4)
```

Since each entry in the dispatch table is 4 bytes long, the kernel finds the address of the service routine to be invoked by first 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.

When the service routine terminates, `system_call( )` gets its return code from `eax` and stores it in the stack location where the User Mode value of the `eax` register has been saved. It then jumps to `ret_from_sys_call( )`, which terminates the execution of the system call handler (see Section 4.7.2 in Chapter 4):

```
movl %eax, 24(%esp)
jmp ret_from_sys_call
```

When the process resumes its execution in User Mode, it will find in `eax` the return code of the system call.

8.2.3 Parameter Passing

Like ordinary functions, system calls often require some input/output parameters, which may consist of actual values (i.e., numbers) or addresses of functions and variables in the address space of the User Mode process. Since the `system_call( )` function is the unique entry point 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 5 before executing the `int $0x80` Assembly 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 parameters (besides the system call number).

The parameters of ordinary functions are passed by writing their values in the active program stack (either the User Mode stack or the Kernel Mode stack). But system call parameters are usually passed to the system call handler in the CPU registers, then copied onto the Kernel Mode stack, since system call service routines are ordinary C functions.

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; moreover, the use of registers makes the structure of the system call handler similar to that of other exception handlers.

However, in order to pass parameters in registers, two conditions must be satisfied:

- The length of each parameter cannot exceed the length of a register, that is 32 bits.[1]
- The number of parameters must not exceed six (including the system call number passed in `eax`), since the Intel Pentium has a very limited number of registers.

[1] We refer as usual to the 32-bit architecture of the Intel 80x86 processors. The discussion in this section does not apply to Compaq's Alpha 64-bit processors.

The first condition is always true since, according to the POSIX standard, large parameters that cannot be stored in a 32-bit register must be passed by specifying their addresses. A typical example is the `settimeofday( )` system call, which must read two 64-bit structures.

However, system calls that have 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 any C 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 six registers used to store system call parameters are, in increasing order: `eax` (for the system call number), `ebx` , `ecx`, `edx`, `esi`, and `edi`. As seen before, `system_call( )` saves 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( )`, followed by the parameter stored in `ebx` (that is, the first parameter of the system call), the parameter stored in `ecx`, and so on (see the Section 4.6.3 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 make use of any parameters, the corresponding service routine needs to know the contents of the CPU registers right before the system call was issued. As an 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 TSS. In these cases, a 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 Section 4.6.4 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.

8.2.4 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 us go back to the `write( )` system call introduced before: the `fd` parameter should be a file descriptor that describes 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 Section 1.5.6 in Chapter 1). If any of these conditions is 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).

Previous Linux kernels performed the first type of checking. But it is quite time-consuming since 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, the Linux 2.2 kernel performs the second type of checking. It is much more efficient because it does not require any scan of the process memory region descriptors. Obviously, it 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 in Linux 2.2 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 shall discuss in Section 8.2.6 later in this chapter how the "Page fault" exception handler succeeds in detecting those bad addresses issued in Kernel Mode that have been 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 have seen 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 any page present in memory without causing a "Page fault" exception!

The check on addresses passed to system calls is performed by the `verify_area( )` function, which acts on two[2] parameters denoted as `addr` and `size`. The function checks the address interval delimited by `addr` and `addr + size - 1`, and is essentially equivalent to the following C function:

[2] A third parameter named `type` specifies whether the system call should read or write the referred memory locations. It is used only in systems having buggy versions of the Intel 80486 microprocessor, in which writing in Kernel Mode to a write-protected page does not generate a page fault. We don't discuss this case further.

```
int verify_area(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a || a + size > current->addr_limit.seg)
        return -EFAULT;
    return 0;
}
```

The function verifies first whether `addr + size`, the highest address to be checked, is larger than 2^{32}-1; since 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 exceeds the value stored in the addr_limit.seg field of current. This field usually has the value PAGE_OFFSET-1 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 invoke system call service routines directly and pass addresses in the kernel data segment to them.

The access_ok macro performs the same check as verify_area( ). It yields 1 if the specified address interval is valid and otherwise.

8.2.5 Accessing the Process Address Space

System call service routines quite 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 us consider one of them, for instance, __get_user_2( ):

```
__get_user_2:
    addl $1, %eax
    jc bad_get_user
    movl %esp, %edx
    andl $0xffffe000, %edx
    cmpl 12(%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 verify_area( ) functions: 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 12 in the process descriptor, 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 2 least significant bytes of edx register while setting the high-order bytes of edx to 0; then it sets a 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 that it writes the value x into the process address space starting from address ptr. Depending on the size of x

(1, 2, or 4 bytes), it invokes the `__put_user_1( )`, `__put_user_2( )`,or `__put_user_4( )` function. Let's consider `__put_user_4( )` for our example this time. The function performs the usual checks on the `ptr` address stored in the `eax` register, then executes a `movl` instruction to write the 4 bytes stored into the `edx` register. The function returns the value in the `eax` register if it succeeds, and `-EFAULT` otherwise.

Several other functions and macros are available to access the process address space in Kernel Mode; they are listed in Table 8-1. 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, then access the process area without making any further checks.

Table 8-1. 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)
`get_user_ret` `__get_user_ret`	Like `get_user`, but returns a specified value on error
`put_user_ret` `__put_user_ret`	Like `put_user`, but returns a specified value on error
`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
`copy_from_user_ret`	Like `copy_from_user`, but returns a specified value on error
`copy_to_user_ret`	Like `copy_to_user`, but returns a specified value on error
`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

8.2.6 Dynamic Address Checking: The Fixup Code

As seen previously, the `verify_area( )` function and the `access_ok` macro make only a coarse check on the validity of linear addresses passed as parameters of a system call. Since they do not ensure that these addresses are included in the process address space, a process could cause a "Page fault" exception by passing a wrong address.

Before describing how the kernel detects this type of error, let us specify the three cases in which "Page fault" exceptions may occur in Kernel Mode:

- 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 is trying to write a read-only page.
- Some kernel function includes 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.
- 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.

These cases must be distinguished by the page fault handler, since the actions to be taken are quite different. In the first case, the handler must allocate and initialize a new page frame (see Section 7.4.3 and Section 7.4.4 in Chapter 7); in the second case, the handler must perform a kernel oops (see Section 7.4.1 in Chapter 7); in the third case, the handler must terminate the system call by returning a proper error code.

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. Let us now explain how the handler distinguishes the remaining two cases.

8.2.6.1 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 ever used to access that 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 be generated by expanding one of the macros. If you add up the code in all these functions and macros, they consist of a fairly small set of addresses.

Therefore, it would not take much effort to put the address of any 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 some 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, *Modules*) includes its own local exception table. This table is automatically generated by the C compiler when building the module image, and it is loaded in memory when the module is inserted in the running kernel.

Each entry of an exception table is an exception_table_entry structure having 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 shall 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. Such instructions are usually defined in the same macro or function that accesses the process address space; sometimes, they are placed by the C compiler in a separate section of the kernel code segment called .fixup.

The search_exception_table( ) function is used to search for a specified address in all exception tables: if the address is included in a table, the function returns the corresponding fixup address; otherwise, it returns 0. Thus the page fault handler do_page_fault( ) executes the following statements:

```
if ((fixup = search_exception_table(regs->eip)) != 0) {
    regs->eip = fixup;
    return;
}
```

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 returned by search_exception_table( ). Then the page fault handler terminates and the interrupted program resumes with execution of the fixup code.

8.2.6.2 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 shall see in Chapter 19, 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 in 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 us consider again the `__get_user_1( )`, `__get_user_2( )`, and `__get_user_4( )` functions mentioned before:

```
_ _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
```

The instructions that access the process address space are those labeled as `1`, `2`, and `3`. The fixup code is common to the three functions and is labeled as `bad_get_user`. Each exception table entry consists simply of two labels. The first one is a numeric label with a `b` suffix to indicate that the label is a "backward" one: in other words, it appears in a previous line of the program. The fixup code at `bad_get_user` returns an `EFAULT` error code to the process that issued the system call.

Let us consider a second example, the `strlen_user(string)` macro. This returns the length of a null-terminated string in the process address space or the value on error. The macro essentially yields the following assembly language instructions:

```
movl $0, %eax
    movl $0x7fffffff, %ecx
    movl %ecx, %edx
    movl string, %edi
0:  repne; scasb
    subl %ecx, %edx
    movl %edx, %eax
1:
.section .fixup,"ax"
2:  movl $0, %eax
    jmp 1b
.previous
.section __ex_table,"a"
    .long 0b, 2b
.previous
```

The ecx and edx registers are initialized with the 0x7fffffff value, which represents the maximum allowed length for the string. The repne; scasb assembly language instructions iteratively scan the string pointed to by the edi register, looking for the value (the end of string \0 character) in eax. Since the ecx register is decremented at each iteration, the eax register will ultimately store 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 in 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 in eax, thus forcing the macro to return a error code instead of the string length, then jumps to the 1 label, which corresponds to the instruction following the macro.

8.3 Wrapper Routines

Although system calls are mainly used by User Mode processes, they can also be invoked by kernel threads, which cannot make use of library functions. In order to simplify the declarations of the corresponding wrapper routines, Linux defines a set of six macros called _syscall0 through _syscall5.

The numbers through 5 in the name of each macro correspond to the number of parameters used by the system call (excluding the system call number). The macros may also be used to simplify the declarations of the wrapper routines in the *libc* standard library; however, they cannot be used to define wrapper routines for system calls having more than five parameters (excluding the system call number) or for system calls that yield nonstandard return values.

Each macro requires exactly 2+2x*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)-125) {
        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 $-126, %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 -125 (the kernel assumes that the largest error code defined in *include/asm-i386/errno.h* is 125). If this is the case, the wrapper routine will store the value of `-eax` in `errno` and return the value -1; otherwise, it will return the value of `eax`.

8.4 Anticipating Linux 2.4

Beside adding a few new system calls, Linux 2.4 does not introduce any change to the system call mechanism of Linux 2.2.

Chapter 10. Process Scheduling

Like any 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 switch 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 10.1 introduces the choices made by Linux to schedule processes in the abstract. Section 10.2 discusses the data structures used to implement scheduling and the corresponding algorithm. Finally, Section 10.3 describes the system calls that affect process scheduling.

10.1 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 selecting a new process to run is called *scheduling policy*.

Linux scheduling is based on the *time-sharing* technique already introduced in Section 5.4.3 in Chapter 5: several processes are allowed to run "concurrently," which means that the CPU time is roughly divided into "slices," one for each runnable process.[1] 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 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 in order to ensure CPU time-sharing.

[1] Recall that stopped and suspended processes cannot be selected by the scheduling algorithm to run on the CPU.

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 denotes how appropriate it is to be assigned to the 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 the 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 are 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 ms. 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. Since 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 strong scheduling requirements. Such processes should never be blocked by lower-priority processes, they should have a short response time and, most important, such response time should have 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 in Linux real-time programs are explicitly recognized as such by the scheduling algorithm, there is no way to distinguish between interactive and batch programs. In order to offer a good response time to interactive applications, Linux (like all Unix kernels) implicitly favors I/O-bound processes over CPU-bound ones.

Programmers may change the scheduling parameters by means of the system calls illustrated in Table 10-1. More details will be given in Section 10.3.

Table 10-1. System Calls Related to Scheduling

System Call	Description
`nice( )`	Change the priority of a conventional process.
`getpriority( )`	Get the maximum priority of a group of conventional processes.
`setpriority( )`	Set the priority of a group of conventional processes.
`sched_getscheduler( )`	Get the scheduling policy of a process.
`sched_setscheduler( )`	Set the scheduling policy and priority of a process.
`sched_getparam( )`	Get the scheduling priority of a process.
`sched_setparam( )`	Set the priority of a process.
`sched_yield( )`	Relinquish the processor voluntarily without blocking.
`sched_get_ priority_min( )`	Get the minimum priority value for a policy.
`sched_get_ priority_max( )`	Get the maximum priority value for a policy.
`sched_rr_get_interval( )`	Get the time quantum value for the Round Robin policy.

Most system calls shown in the table apply to real-time processes, thus allowing users to develop real-time applications. However, Linux does not support the most demanding real-time applications because its kernel is nonpreemptive (see Section 10.2.5).

10.1.1 Process Preemption

As mentioned in the first chapter, Linux processes are *preemptive*. If 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 may also be preempted when its time quantum expires. As mentioned in Section 5.4.3 in Chapter 5, when this occurs, the `need_resched` field of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.

For instance, let us consider a scenario in which only two programs—a text editor and a compiler—are being executed. The text editor is an interactive program, therefore it has a higher dynamic priority than the compiler. Nevertheless, it is often suspended, since 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 (that is, the compiler), and hence it sets the `need_resched` field 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 task 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, since it remains in the `TASK_RUNNING` state; it simply no longer uses the CPU.

Some real-time operating systems feature preemptive kernels, which means that a process running in Kernel Mode can be interrupted after any instruction, just as it can in User Mode. The Linux kernel is not preemptive, which means that a process can be preempted only while running in User Mode; nonpreemptive kernel design is much simpler, since most synchronization problems involving the kernel data structures are easily avoided (see Section 11.2.1 in Chapter 11).

10.1.2 How Long Must a Quantum Last?

The quantum duration is critical for system performances: it should be neither too long nor too short.

If the quantum duration is too short, the system overhead caused by task switches becomes excessively high. For instance, suppose that a task switch requires 10 milliseconds; if the quantum is also set to 10 milliseconds, then at least 50% of the CPU cycles will be dedicated to task switch.[2]

[2] Actually, things could be much worse than this; for example, if the time required for task switch is counted in the process quantum, all CPU time will be devoted to task switch and no process can progress toward its termination. Anyway, you got the point.

If the 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 Section 10.1.1 earlier in this chapter, interactive processes have a relatively high priority, therefore they quickly preempt the batch processes, no matter how long the quantum duration is.

In some cases, a quantum duration that is too long degrades the responsiveness of the system. For instance, suppose that two users concurrently enter two commands at the respective shell prompts; one command is CPU-bound, while the other is an interactive application. Both shells fork a new process and delegate the execution of the user's command to it; moreover, suppose that such new processes have the same priority initially (Linux does not know in advance if an executed program is batch or interactive). Now, if the scheduler selects the CPU-bound process to run, the other process could wait for a whole time quantum before starting its execution. Therefore, if such duration is long, the system could appear to be unresponsive to the user that launched it.

The choice of 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.

10.2 The Scheduling Algorithm

The Linux scheduling algorithm works by dividing the CPU time into *epochs* . In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins. In general, different processes have different time quantum durations. The time quantum value is the maximum CPU time portion assigned to the process in that epoch. When a process has exhausted its time quantum, it is preempted and replaced by another runnable process. Of course, a process can be selected several times from the scheduler in the same epoch, as long as its quantum has not been exhausted—for instance, if it suspends itself to wait for I/O, it preserves some of its time quantum and can be selected again during the same epoch. The epoch ends when all runnable processes have exhausted their quantum; in this case, the scheduler algorithm recomputes the time-quantum durations of all processes and a new epoch begins.

Each process has a *base time quantum*: it is the time-quantum value assigned by the scheduler to the process if it has exhausted its quantum in the previous epoch. The users can change the base time quantum of their processes by using the `nice( )` and `setpriority( )` system calls (see Section 10.3 later in this chapter). A new process always inherits the base time quantum of its parent.

The `INIT_TASK` macro sets the value of the base time quantum of process (*swapper*) to `DEF_PRIORITY`; that macro is defined as follows:

```
#define DEF_PRIORITY (20*HZ/100)
```

Since `HZ`, which denotes the frequency of timer interrupts, is set to 100 for IBM PCs (see Section 5.1.3 in Chapter 5), the value of `DEF_PRIORITY` is 20 ticks, that is, about 210 ms.

Users rarely change the base time quantum of their processes, so `DEF_PRIORITY` also denotes the base time quantum of most processes in the system.

In order to select a process to run, the Linux scheduler must consider the priority of each process. Actually, there are two kinds of priority:

Static priority

This kind is assigned by the users to real-time processes and ranges from 1 to 99. It is never changed by the scheduler.

Dynamic priority

This kind applies only to conventional processes; it is essentially the sum of the base time quantum (which is therefore also called the *base priority* of the process) and of the number of ticks of CPU time left to the process before its quantum expires in the current epoch.

Of course, the static priority of a real-time process is always higher than the dynamic priority of a conventional one: the scheduler will start running conventional processes only when there is no real-time process in a `TASK_RUNNING` state.

10.2.1 Data Structures Used by the Scheduler

We recall from Section 3.1 in Chapter 3 that the process list links together all process descriptors, while the runqueue list links together the process descriptors of all runnable processes—that is, of those in a `TASK_RUNNING` state. In both cases, the `init_task` process descriptor plays the role of list header.

Each process descriptor includes several fields related to scheduling:

`need_resched`

A flag checked by `ret_from_intr( )` to decide whether to invoke the `schedule( )` function (see Section 4.7.1 in Chapter 4).

`policy`

The scheduling class. The values permitted are:

`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 will continue to use the CPU as long as it wishes, even if other real-time processes having 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_OTHER

A conventional, time-shared process.

The policy field also encodes a SCHED_YIELD binary flag. This flag is set when the process invokes the sched_ yield( ) system call (a way of voluntarily relinquishing the processor without the need to start an I/O operation or go to sleep; see Section 10.3.3). The scheduler puts the process descriptor at the bottom of the runqueue list (see Section 10.3).

rt_priority

The static priority of a real-time process. Conventional processes do not make use of this field.

priority

The base time quantum (or base priority) of the process.

counter

The number of ticks of CPU time left to the process before its quantum expires; when a new epoch begins, this field contains the time-quantum duration of the process. Recall that the update_process_times( ) function decrements the counter field of the current process by 1 at every tick.

When a new process is created, do_fork( ) sets the counter field of both current (the parent) and p (the child) processes in the following way:

```
current->counter >>= 1;
p->counter = current->counter;
```

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 since the kernel does not reward forks. Similarly, a user cannot hog an unfair share of the processor by starting lots of 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.

Notice that the priority and counter fields play different roles for the various kinds of processes. For conventional processes, they are used both to implement time-sharing and to compute the process dynamic priority. For SCHED_RR real-time processes, they are used only to implement time-sharing. Finally, for SCHED_FIFO real-time processes, they are not used at all, because the scheduling algorithm regards the quantum duration as unlimited.

10.2.2 The schedule( ) Function

schedule( ) 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 way, by several kernel routines.

10.2.2.1 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 if the resource is available; if not, goes to step 2
5. Once the resource is available, removes current from the wait queue

As can be seen, 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 again checked.

You may have noticed that these steps are similar to those performed by the sleep_on( ) and interruptible_sleep_on( ) functions described in Section 3.1.4 in Chapter 3. However, the functions we discuss here immediately remove the process from the wait queue as soon as it is woken up.

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 need_resched field and, if necessary, invokes schedule( ) to voluntarily relinquish the CPU.

10.2.2.2 Lazy invocation

The scheduler can also be invoked in a lazy way by setting the need_resched field of current to 1. Since a check on the value of this field is always made before resuming the execution of a User Mode process (see Section 4.7 in Chapter 4), schedule( ) will definitely be invoked at some close future time.

Lazy invocation of the scheduler is performed in the following cases:

- When current has used up its quantum of CPU time; this is done by the update_process_times( ) function.

- When a process is woken up and its priority is higher than that of the current process; this task is performed by the `reschedule_idle( )` function, which is invoked by the `wake_up_process( )` function (see Section 3.1.2 in Chapter 3):

-
  ```
   if (goodness(current, p) > goodness(current, current))
      current->need_resched = 1;
  ```

  (The `goodness( )` function will be described later in Section 10.2.3)

- When a `sched_setscheduler( )` or `sched_ yield( )` system call is issued (see Section 10.3 later in this chapter).

10.2.2.3 Actions performed by schedule( )

Before actually scheduling a process, the `schedule( )` function starts by running the functions left by other kernel control paths in various queues. The function invokes `run_task_queue( )` on the `tq _scheduler` task queue. Linux puts a function in that task queue when it must defer its execution until the next `schedule( )` invocation:

```
run_task_queue(&tq_scheduler);
```

The function then executes all active unmasked bottom halves. These are usually present to perform tasks requested by device drivers (see Section 4.6.6 in Chapter 4):

```
if (bh_active & bh_mask)
    do_bottom_half( );
```

Now comes the actual scheduling, and therefore a potential process switch.

The value of `current` is saved in the `prev` local variable and the `need_resched` field of `prev` is set to 0. The key outcome of the function is to set another local variable called `next` so that it points to the descriptor of the process selected to replace `prev`.

First, a check is made to determine whether `prev` is a Round Robin real-time process (`policy` field set to `SCHED_RR`) that has exhausted its quantum. If so, `schedule( )` assigns a new quantum to `prev` and puts it at the bottom of the runqueue list:

```
if (!prev->counter && prev->policy == SCHED_RR) {
    prev->counter = prev->priority;
    move_last_runqueue(prev);
}
```

Now `schedule( )` examines the state of `prev`. If it has nonblocked pending signals and its state is `TASK_INTERRUPTIBLE`, the function wakes up the process as follows. 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_INTERRUPTIBLE &&
    signal_pending(prev))
    prev->state = TASK_RUNNING;
```

If prev is not in the TASK_RUNNING state, schedule( ) was directly invoked by the process itself because it had to wait on some external resource; therefore, prev must be removed from the runqueue list:

```
if (prev->state != TASK_RUNNING)
    del_from_runqueue(prev);
```

Next, schedule( ) must select the process to be executed in the next time quantum. To that end, the function scans the runqueue list. It starts from the process referenced by the next_run field of init_task, which is the descriptor of process (*swapper*). The objective is to store in next the process descriptor pointer of the highest priority process. In order to do this, next is initialized to the first runnable process to be checked, and c is initialized to its "goodness" (see Section 10.2.3):

```
if (prev->state == TASK_RUNNING) {
    next = prev;
    if (prev->policy & SCHED_YIELD) {
        prev->policy &= ~SCHED_YIELD;
        c = 0;
    } else
        c = goodness(prev, prev);
} else {
    c = -1000;
    next = &init_task;
}
```

If the SCHED_YIELD flag of prev->policy is set, prev has voluntarily relinquished the CPU by issuing a sched_ yield( ) system call. In this case, the function assigns a zero goodness to it.

Now schedule( ) repeatedly invokes the goodness( ) function on the runnable processes to determine the best candidate:

```
p = init_task.next_run;
while (p != &init_task) {
    weight = goodness(prev, p);
    if (weight > c) {
        c = weight;
        next = p;
    }
    p = p->next_run;
}
```

The while loop selects the first process in the runqueue having maximum weight. If the previous process is runnable, it is preferred with respect to other runnable processes having the same weight.

Notice that if the runqueue list is empty (no runnable process exists except for *swapper*), the cycle is not entered and next points to init_task. Moreover, if all processes in the runqueue list have a priority lesser than or equal to the priority of prev, no process switch will take place and the old process will continue to be executed.

A further check must be made at the exit of the loop to determine whether c is 0. This occurs only when all the processes in the runqueue list have exhausted their quantum, that is, all of them have a zero `counter` field. When this happens, a new epoch begins, therefore `schedule( )` assigns to all existing processes (not only to the `TASK_RUNNING` ones) a fresh quantum, whose duration is the sum of the `priority` value plus half the `counter` value:

```
if (!c) {
    for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
}
```

In this way, suspended or stopped processes have their dynamic priorities periodically increased. As stated earlier, the rationale for increasing the `counter` value of suspended or stopped processes is to give preference to I/O-bound processes. However, even after an infinite number of increases, the value of `counter` can never become larger than twice[3] the `priority` value.

Now comes the concluding part of `schedule( )`: if a process other than `prev` has been selected, a process switch must take place. Before performing it, however, the `context_swtch` field of `kstat` is increased by 1 to update the statistics maintained by the kernel:

```
if (prev != next) {
    kstat.context_swtch++;
    switch_to(prev,next);
}
return;
```

Notice that the `return` statement that exits from `schedule( )` will not be performed right away by the `next` process but at a later time by the `prev` one when the scheduler selects it again for execution.

10.2.3 How Good Is a Runnable Process?

The heart of the scheduling algorithm includes identifying the best candidate among all processes in the runqueue list. This is what the `goodness( )` function does. It receives as input parameters `prev` (the descriptor pointer of the previously running process) and `p` (the descriptor pointer of the process to evaluate). The integer value `c` returned by `goodness( )` measures the "goodness" of `p` and has the following meanings:

c = -1000

`p` must never be selected; this value is returned when the runqueue list contains only `init_task`.

c = 0

`p` has exhausted its quantum. Unless `p` is the first process in the runqueue list and all runnable processes have also exhausted their quantum, it will not be selected for execution.

[3] Assume both `priority` and `counter` equal to P; then the geometric series Px (1 + $^1/_2$ + $^1/_4$ + $^1/_8$ + . . .) converges to 2 xP.

0 < *c* < 1000

p is a conventional process that has not exhausted its quantum; a higher value of c denotes a higher level of goodness.

c >= 1000

p is a real-time process; a higher value of c denotes a higher level of goodness.

The goodness( ) function is equivalent to:

```
if (p->policy != SCHED_OTHER)
       return 1000 + p->rt_priority;
if (p->counter == 0)
       return 0;
if (p->mm == prev->mm)
       return p->counter + p->priority + 1;
return p->counter + p->priority;
```

If the process is real-time, its goodness is set to at least 1000. If it is a conventional process that has exhausted its quantum, its goodness is set to 0; otherwise, it is set to p->counter + p->priority.

A small bonus is given to p if it shares the address space with prev (i.e., if their process descriptors' mm fields point to the same memory descriptor). The rationale for this bonus is that if p runs right after prev, it will use the same page tables, hence the same memory; some of the valuable data may still be in the hardware cache.

10.2.4 The Linux/SMP Scheduler

The Linux scheduler must be slightly modified in order to support the symmetric multiprocessor (SMP) architecture. Actually, each processor runs the schedule( ) function on its own, but processors must exchange information in order to boost system performance.

When the scheduler computes the goodness of a runnable process, it should consider whether that process was previously running on the same CPU or on another one. A process that was running on the same CPU is always preferred, since the hardware cache of the CPU could still include useful data. This rule helps in reducing the number of cache misses.

Let us suppose, however, that CPU 1 is running a process when a second, higher-priority process that was last running on CPU 2 becomes runnable. Now the kernel is faced with an interesting dilemma: should it immediately execute the higher-priority process on CPU 1, or should it defer that process's execution until CPU 2 becomes available? In the former case, hardware caches contents are discarded; in the latter case, parallelism of the SMP architecture may not be fully exploited when CPU 2 is running the idle process (*swapper*).

In order to achieve good system performance, Linux/SMP adopts an empirical rule to solve the dilemma. The adopted choice is always a compromise, and the trade-off mainly depends on the size of the hardware caches integrated into each CPU: the larger the CPU cache is, the more convenient it is to keep a process bound on that CPU.

10.2.4.1 Linux/SMP scheduler data structures

An `aligned_data` table includes one data structure for each processor, which is used mainly to obtain the descriptors of current processes quickly. Each element is filled by every invocation of the `schedule( )` function and has the following structure:

```
struct schedule_data {
    struct task_struct * curr;
    unsigned long last_schedule;
};
```

The `curr` field points to the descriptor of the process running on the corresponding CPU, while `last_schedule` specifies when `schedule( )` selected `curr` as the running process.

Several SMP-related fields are included in the process descriptor. In particular, the `avg_slice` field keeps track of the average quantum duration of the process, and the `processor` field stores the logical identifier of the last CPU that executed it.

The `cacheflush_time` variable contains a rough estimate of the minimal number of CPU cycles it takes to entirely overwrite the hardware cache content. It is initialized by the `smp_tune_scheduling( )` function to:

$$\left\lfloor \frac{\textit{cache size in KB}}{5000} \times \textit{cpu frequency in kHz} \right\rfloor$$

Intel Pentium processors have a hardware cache of 8 KB, so their `cacheflush_time` is initialized to a few hundred CPU cycles, that is, a few microseconds. Recent Intel processors have larger hardware caches, and therefore the minimal cache flush time could range from 50 to 100 microseconds.

As we shall see later, if `cacheflush_time` is greater than the average time slice of some currently running process, no process preemption is performed because it is convenient in this case to bind processes to the processors that last executed them.

10.2.4.2 The schedule( ) function

When the `schedule( )` function is executed on an SMP system, it carries out the following operations:

1. Performs the initial part of `schedule( )` as usual.
2. Stores the logical identifier of the executing processor in the `this_cpu` local variable; such value is read from the `processor` field of `prev` (that is, of the process to be replaced).
3. Initializes the `sched_data` local variable so that it points to the `schedule_data` structure of the `this_cpu` CPU.
4. Invokes `goodness( )` repeatedly to select the new process to be executed; this function also examines the `processor` field of the processes and gives a consistent bonus (`PROC_CHANGE_PENALTY`, usually 15) to the process that was last executed on the `this_cpu` CPU.
5. If needed, recomputes process dynamic priorities as usual.
6. Sets `sched_data->curr` to `next`.

7. Sets `next->has_cpu` to 1 and `next->processor` to `this_cpu`.
8. Stores the current Time Stamp Counter value in the `t` local variable.
9. Stores the last time slice duration of `prev` in the `this_slice` local variable; this value is the difference between `t` and `sched_data->last_schedule`.
10. Sets `sched_data->last_schedule` to `t`.
11. Sets the `avg_slice` field of `prev` to (`prev->avg_slice+this_slice`)/2; in other words, updates the average.
12. Performs the context switch.
13. When the kernel returns here, the original previous process has been selected again by the scheduler; the `prev` local variable now refers to the process that has just been replaced. If `prev` is still runnable and it is not the idle task of this CPU, invokes the `reschedule_idle( )` function on it (see the next section).
14. Sets the `has_cpu` field of `prev` to 0.

10.2.4.3 The reschedule_idle( ) function

The `reschedule_idle( )` function is invoked when a process `p` becomes runnable (see Section 10.2.2). On an SMP system, the function determines whether the process should preempt the current process of some CPU. It performs the following operations:

1. If `p` is a real-time process, always attempts to perform preemption: go to step 3.
2. Returns immediately (does not attempt to preempt) if there is a CPU whose current process satisfies both of the following conditions:[4]
    - `cacheflush_time` is greater than the average time slice of the current process. If this is true, the process is not dirtying the cache significantly.
    - Both `p` and the current process need the global kernel lock (see Section 11.4.6 in Chapter 11) in order to access some critical kernel data structure. This check is performed because replacing a process holding the lock with another one that needs it is not fruitful.
3. If the `p->processor` CPU (the one on which `p` was last running) is idle, selects it.
4. Otherwise, computes the difference:

    ```
    goodness(tsk, p) - goodness(tsk, tsk)
    ```

    for each task `tsk` running on some CPU and selects the CPU for which the difference is greatest, provided it is a positive value.

5. If CPU has been selected, sets the `need_resched` field of the corresponding running process and sends a "reschedule" message to that processor (see Section 11.4.7 in Chapter 11).

10.2.5 Performance of the Scheduling Algorithm

The scheduling algorithm of Linux is both self-contained and relatively easy to follow. For that reason, many kernel hackers love to try to make improvements. However, the scheduler is a rather mysterious component of the kernel. While you can change its performance significantly by modifying just a few key parameters, there is usually no theoretical support to

[4] These conditions look like voodoo magic; perhaps, they are empirical rules that make the SMP scheduler work better.

justify the results obtained. Furthermore, you can't be sure that the positive (or negative) results obtained will continue to hold when the mix of requests submitted by the users (real-time, interactive, I/O-bound, background, etc.) varies significantly. Actually, for almost every proposed scheduling strategy, it is possible to derive an artificial mix of requests that yields poor system performances.

Let us try to outline some pitfalls of the Linux scheduler. As it will turn out, some of these limitations become significant on large systems with many users. On a single workstation that is running a few tens of processes at a time, the Linux scheduler is quite efficient. Since Linux was born on an Intel 80386 and continues to be most popular in the PC world, we consider the current Linux scheduler quite appropriate.

10.2.5.1 The algorithm does not scale well

If the number of existing processes is very large, it is inefficient to recompute all dynamic priorities at once.

In old traditional Unix kernels, the dynamic priorities were recomputed every second, thus the problem was even worse. Linux tries instead to minimize the overhead of the scheduler. Priorities are recomputed only when all runnable processes have exhausted their time quantum. Therefore, when the number of processes is large, the recomputation phase is more expensive but is executed less frequently.

This simple approach has the disadvantage that when the number of runnable processes is very large, I/O-bound processes are seldom boosted, and therefore interactive applications have a longer response time.

10.2.5.2 The predefined quantum is too large for high system loads

The system responsiveness experienced by users depends heavily on the *system load*, which is the average number of processes that are runnable, and hence waiting for CPU time.[5]

[5] The `uptime` program returns the system load for the past 1, 5, and 15 minutes. The same information can be obtained by reading the */proc/loadavg*file.

As mentioned before, system responsiveness depends also on the average time-quantum duration of the runnable processes. In Linux, the predefined time quantum appears to be too large for high-end machines having a very high expected system load.

10.2.5.3 I/O-bound process boosting strategy is not optimal

The preference for I/O-bound processes is a good strategy to ensure a short response time for interactive programs, but it is not perfect. Indeed, some batch programs with almost no user interaction are I/O-bound. For instance, consider a database search engine that must typically read lots of data from the hard disk or a network application that must collect data from a remote host on a slow link. Even if these kinds of processes do not need a short response time, they are boosted by the scheduling algorithm.

On the other hand, interactive programs that are also CPU-bound may appear unresponsive to the users, since the increment of dynamic priority due to I/O blocking operations may not compensate for the decrement due to CPU usage.

10.2.5.4 Support for real-time applications is weak

As stated in the first chapter, nonpreemptive kernels are not well suited for real-time applications, since processes may spend several milliseconds in Kernel Mode while handling an interrupt or exception. During this time, a real-time process that becomes runnable cannot be resumed. This is unacceptable for real-time applications, which require predictable and low response times.[6]

[6] The Linux kernel has been modified in several ways so it can handle a few hard real-time jobs if they remain short. Basically, hardware interrupts are trapped and kernel execution is monitored by a kind of "superkernel." These changes do not make Linux a true real-time system, though.

Future versions of Linux will likely address this problem, either by implementing SVR4's "fixed preemption points" or by making the kernel fully preemptive.

However, kernel preemption is just one of several necessary conditions for implementing an effective real-time scheduler. Several other issues must be considered. For instance, real-time processes often must use resources also needed by conventional processes. A real-time process may thus end up waiting until a lower-priority process releases some resource. This phenomenon is called *priority inversion*. Moreover, a real-time process could require a kernel service that is granted on behalf of another lower-priority process (for example, a kernel thread). This phenomenon is called *hidden scheduling*. An effective real-time scheduler should address and resolve such problems.

10.3 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.

10.3.1 The nice( ) System Call

The `nice( )`[7] system call allows processes to change their base priority. The integer value contained in the `increment` parameter is used to modify the `priority` 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.

[7] Since this system call is usually invoked to lower the priority of a process, users who invoke it for their processes are "nice" toward other users.

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 decrements.

The function starts by copying the value of `increment` into the `newprio` local variable. In the case of a negative increment, the function invokes the `capable( )` function to verify whether the process has a `CAP_SYS_NICE` capability. We shall discuss that function, together with the notion of capability, in Chapter 19. If the user turns out to have the capability required to change priorities, `sys_nice( )` changes the sign of `newprio` and it sets the `increase` local flag:

```
increase = 0
newprio = increment;
if (increment < 0) {
    if (!capable(CAP_SYS_NICE))
        return -EPERM;
    newprio = -increment;
    increase = 1;
}
```

If `newprio` has a value larger than 40, the function trims it down to 40. At this point, the `newprio` local variable may have any value included from to 40, inclusive. The value is then converted according to the priority scale used by the scheduling algorithm. Since the highest base priority allowed is 2 x `DEF_PRIORITY`, the new value is:

$$\lfloor (\text{newprio} \times 2 \times \text{DEF\_PRIORITY})/40 + 0.5 \rfloor$$

The resulting value is copied into `increment` with the proper sign:

```
if (newprio > 40)
    newprio = 40;
newprio = (newprio * DEF_PRIORITY + 10) / 20;
increment = newprio;
if (increase)
    increment = -increment;
```

Since `newprio` is an integer variable, the expression in the code is equivalent to the formula shown earlier.

The function then sets the final value of `priority` by subtracting the value of `increment` from it. However, the final base priority of the process cannot be smaller than 1 or larger than 2 x `DEF_PRIORITY`:

```
if (current->priority - increment < 1)
    current->priority = 1;
else if (current->priority > DEF_PRIORITY*2)
    current->priority = DEF_PRIORITY*2;
else
    current->priority -= increment;
return 0;
```

A `niced` process changes over time like any other process, getting extra priority if necessary or dropping back in deference to other processes.

10.3.2 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 plus the highest base priority among all processes in a given group; `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

Identifies the group of processes; it can assume one of the following values:

PRIO_PROCESS

Select the processes according to their process ID (pid field of the process descriptor).

PRIO_PGRP

Select the processes according to their group ID (pgrp field of the process descriptor).

PRIO_USER

Select the processes according to their user ID (uid field of the process descriptor).

who

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 +20 (minimum 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 have seen in Chapter 8, system calls return a negative value only if some error occurred. For that reason, getpriority( ) does not return a normal nice value ranging between -20 and 20, but rather a nonnegative value ranging between and 40.

10.3.3 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 in order to modify the values of the rt_priority and policy process descriptor fields of any process, including itself.

10.3.3.1 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 will be retrieved. On success, the system call returns the policy for the process: SCHED_FIFO , `SCHED_RR`, or `SCHED_OTHER`. The corresponding `sys_sched_getscheduler( )` service routine invokes `find_task_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( )` function checks whether the scheduling policy specified by the `policy` parameter and the new static 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 executes the following statements:

```
p->policy = policy;
p->rt_priority = param->sched_priority;
if (p->next_run)
    move_first_runqueue(p);
current->need_resched = 1;
```

10.3.3.2 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( )`: it differs from the latter by not letting the caller set the `policy` field's value.[8] The corresponding `sys_sched_setparam( )` service routine is almost identical to `sys_sched_setscheduler( )`, but the policy of the affected process is never changed.

[8] This anomaly is caused by a specific requirement of the POSIX standard.

10.3.3.3 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 at the end of the runqueue list. In this way, other processes having the same dynamic priority will have a chance to run. The call is used mainly by `SCHED_FIFO` processes.

The corresponding `sys_sched_ yield( )` service routine executes these statements:

```
if (current->policy == SCHED_OTHER)
    current->policy |= SCHED_YIELD;
current->need_resched = 1;
move_last_runqueue(current);
```

Notice that the `SCHED_YIELD` field is set in the `policy` field of the process descriptor only if the process is a conventional `SCHED_OTHER` process. As a result, the next invocation of `schedule( )` will view this process as one that has exhausted its time quantum (see how `schedule( )` handles the `SCHED_YIELD` field).

10.3.3.4 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, otherwise.

The `sys_sched_get_priority_max( )` service routine returns 99 (the highest priority) if `current` is a real-time process, otherwise.

10.3.3.5 The sched_rr_ get_interval( ) system call

The `sched_rr_get_interval( )` system call should get the round robin time quantum for the named real-time process.

The corresponding `sys_sched_rr_get_interval( )` service routine does not operate as expected, since it always returns a 150-millisecond value in the `timespec` structure pointed to by `tp`. This system call remains effectively unimplemented in Linux.

10.4 Anticipating Linux 2.4

Linux 2.4 introduces a subtle optimization concerning TLB flushing for kernel threads and zombie processes. As a result, the active Page Global Directory is set by the `schedule( )` function rather than by the `switch_to` macro.

The Linux 2.4 scheduling algorithm for SMP machines has been improved and simplified. Whenever a new process becomes runnable, the kernel checks whether the preferred CPU of the process, that is, the CPU on which it was last running, is idle; in this case, the kernel assigns the process to that CPU. Otherwise, the kernel assigns the process to another idle CPU, if any. If all CPUs are busy, the kernel checks whether the process has enough priority to preempt the process running on the preferred CPU. If not, the kernel tries to preempt some other CPU only if the new runnable process is real-time or if it has short average time slices compared to the hardware cache rewriting time. (Roughly, preemption occurs if the new runnable process is interactive and the preferred CPU will not reschedule shortly.)

Chapter 11. 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 Section 1.6 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 four basic synchronization techniques implemented by the kernel and illustrate how they are applied by means of examples.

The next two sections deal with the extension of the Linux kernel to multiprocessor architectures. The first describes some hardware features of the Symmetric Multiprocessor (SMP) architecture, while the second discusses additional mutual exclusion techniques adopted by the SMP version of the Linux kernel.

11.1 Kernel Control Paths

As we said, kernel functions are executed following a request that may be issued in two possible ways:

- A process executing in User Mode causes an exception, for instance by executing an `int 0x80` assembly language instruction.
- An external device sends a signal to a Programmable Interrupt Controller by using an IRQ line, and the corresponding interrupt is enabled.

The sequence of instructions executed in Kernel Mode to handle a kernel request is denoted as *kernel control path* : when a User Mode process issues a system call request, for instance, the first instructions of the corresponding kernel control path are those included in the initial part of the `system_call( )` function, while the last instructions are those included in the `ret_from_sys_call( )` function.

In Section 4.3 in Chapter 4, a kernel control path was defined as a sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt. Kernel control paths play a role similar to that of processes, except that they are much more rudimentary: first, no descriptor of any kind is attached to them; second, they are not scheduled through a single function, but rather by inserting sequences of instructions that stop or resume the paths into the kernel code.

In the simplest cases, 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 kernel control paths:

- A context switch occurs. As we have seen in Chapter 10, a context switch can occur only when the `schedule( )` function is invoked.

- An interrupt occurs while the CPU is running a kernel control path with interrupts enabled. In this case, the first kernel control path is left unfinished and the CPU starts processing another kernel control path to handle the interrupt.

It is important to interleave kernel control paths in order to implement multiprocessing. In addition, as already noticed in Section 4.3 in Chapter 4, interleaving improves the throughput of programmable interrupt controllers and device controllers.

While interleaving kernel control paths, special care must be applied to data structures that contain several related member variables, for instance, a buffer and an integer indicating its length. All statements affecting such a data structure must be put into a single critical section, otherwise, it is in danger of being corrupted.

11.2 Synchronization Techniques

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 some computation depends on how two or more interleaved kernel control paths are nested. A critical region is any section of code that should be completely executed by each kernel control path that begins it, before another kernel control path can enter it.

We now examine how kernel control paths can be interleaved while avoiding race conditions among shared data. We'll distinguish four broad types of synchronization techniques:

- Nonpreemptability of processes in Kernel Mode
- Atomic operations
- Interrupt disabling
- Locking

11.2.1 Nonpreemptability of Processes in Kernel Mode

As already pointed out, the Linux kernel is not preemptive, that is, a running process cannot be preempted (replaced by a higher-priority process) while it remains in Kernel Mode. In particular, the following assertions always hold in Linux:

- No process running in Kernel Mode may be replaced by another process, except when the former voluntarily relinquishes control of the CPU.[1]

[1] Of course, all context switches are performed in Kernel Mode. However, a context switch may occur only when the current process is going to return in User Mode.

- Interrupt or exception handling can interrupt a process running in Kernel Mode; however, when the interrupt handler terminates, the kernel control path of the process is resumed.
- A kernel control path performing interrupt or exception handling can be interrupted only by another control path performing interrupt or exception handling.

Thanks to the above assertions, kernel control paths dealing with nonblocking system calls are atomic with respect to other control paths started by system calls. This simplifies the implementation of many kernel functions: any kernel data structures that are not updated by interrupt or exception handlers can be safely accessed. However, if a process in Kernel Mode voluntarily relinquishes the CPU, 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 all previously accessed data structures that could be changed. The change could be caused by a different kernel control path, possibly running the same code on behalf of a separate process.

11.2.2 Atomic Operations

The easiest way to prevent race conditions is by ensuring that an operation is atomic at the chip level: the operation must be executed in a single instruction. These very small atomic operations can be found at the base of other, more flexible mechanisms to create critical sections.

Thus, an *atomic operation* is something that can be performed by executing a single assembly language instruction in an "atomic" way, that is, without being interrupted in the middle.

Let's review Intel 80x86 instructions according to that classification:

- Assembly language instructions that make zero or one 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, naturally, never happens in a uniprocessor system, because all memory accesses are made by the same processor.
- 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 a single, atomic instruction for an operation like `a=a+1` or even for `a++`. Thus, the Linux kernel provides special functions (see Table 11-1) that it implements as single, atomic assembly language instructions; on multiprocessor systems each such instruction is prefixed by a `lock` byte.

Table 11-1. Atomic Operations in C

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_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 non-null, otherwise.
atomic_inc_and_test_greater_zero(v)	Add 1 to *v and return 1 if the result is positive, otherwise.
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.

11.2.3 Interrupt Disabling

For any section of code too large to be defined as an atomic operation, more complicated means of providing critical sections are needed. To ensure that no window is left open for a race condition to slip in, even a window one instruction long, these critical sections always have an atomic operation at their base.

Interrupt disabling is one of the key mechanisms used to ensure that a sequence of kernel statements is operated 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.

However, interrupt disabling alone does not always prevent kernel control path interleaving. Indeed, a kernel control path could raise a "Page fault" exception, which in turn could suspend the current process (and thus the corresponding kernel control path). Or again, a kernel control path could directly invoke the schedule( ) function. This happens during most I/O disk operations because they are potentially blocking, that is, they may force the process to sleep until the I/O operation completes. Therefore, the kernel must never execute a blocking operation when interrupts are disabled, since the system could freeze.

Interrupts can be disabled by means of the cli assembly language instruction, which is yielded by the _ _cli( ) and cli( ) macros. Interrupts can be enabled by means of the sti assembly language instruction, which is yielded by the __sti( ) and sti( ) macros. On a uniprocessor system cli( ) is equivalent to __cli( ) and sti( ) is equivalent to __sti( ); however, as we shall see later in this chapter, these macros are quite different on a multiprocessor system.

When the kernel enters a critical section, it clears the IF flag of the eflags register in order to disable interrupts. But at the end of the critical section, the kernel can't simply set the flag again. Interrupts can execute in nested fashion, so the kernel does not know what the IF flag was before the current control path executed. Each control path must therefore save the old setting of the flag and restore that setting at the end.

In order to save the eflags content, the kernel uses the __save_flags macro; on a uniprocessor system it is identical to the save_flags macro. In order to restore the eflags content, the kernel uses the _ _restore_flags and (on a uniprocessor system) restore_flags macros. Typically, these macros are used in the following way:

```
__save_flags(old);
__cli(  );
[...]
__restore_flags(old);
```

The __save_flags macro copies the content of the eflags register into the old local variable; the IF flag is then cleared by __cli( ). At the end of the critical region, the __restore_flags macro restores the original content of eflags; therefore, interrupts are enabled only if they were enabled before this control path issued the __cli( ) macro.

Linux offers several additional synchronization macros that are important on a multiprocessor system (see Section 11.4.2 later in this chapter) but are somewhat redundant on a uniprocessor system (see Table 11-2). Notice that some functions do not perform any visible operation. They just act as "barriers" for the gcc compiler, since they prevent the compiler from optimizing the code by moving around assembly language instructions. The lck parameter is always ignored.

Table 11-2. Interrupt Disabling/Enabling Macros on a Uniprocessor System

Macro	Description
spin_lock_init(lck)	No operation
spin_lock(lck)	No operation
spin_unlock(lck)	No operation
spin_unlock_wait(lck)	No operation
spin_trylock(lck)	Return always 1
spin_lock_irq(lck)	_ _cli( )
spin_unlock_irq(lck)	_ _sti( )
spin_lock_irqsave(lck, flags)	_ _save_flags(flags); _ _cli( )
spin_unlock_irqrestore(lck, flags)	_ _restore_flags(flags)
read_lock_irq(lck)	_ _cli( )
read_unlock_irq(lck)	_ _sti( )
read_lock_irqsave(lck, flags)	_ _save_flags(flags); _ _cli( )
read_unlock_irqrestore(lck, flags)	_ _restore_flags(flags)
write_lock_irq(lck)	_ _cli( )
write_unlock_irq(lck)	_ _sti( )
write_lock_irqsave(lck, flags)	_ _save_flags(flags); _ _cli( )
write_unlock_irqrestore(lck, flags)	_ _restore_flags(flags)

Let us recall a few examples of how these macros are used in functions introduced in previous chapters:

- The add_wait_queue( ) and remove_wait_queue( ) functions protect the wait queue list with the write_lock_irqsave( ) and write_unlock_irqrestore( ) functions.
- The setup_x86_irq( ) adds a new interrupt handler for a specific IRQ; the spin_lock_irqsave( ) and spin_unlock_irqrestore( ) functions are used to protect the corresponding list of handlers.

- The `run_timer_list( )` function protects the dynamic timer data structures with the `spin_lock_irq( )` and `spin_unlock_irq( )` functions.
- The `handle_signal( )` function protects the `blocked` field of `current` with the `spin_lock_irq( )` and `spin_unlock_irq( )` functions.

Because of its simplicity, interrupt disabling is widely used by kernel functions for implementing critical regions. Clearly, the critical regions obtained by interrupt disabling must be short, because any kind of communication between the I/O device controllers and the CPU is blocked when the kernel enters one. Longer critical regions should be implemented by means of locking.

11.2.4 Locking Through Kernel Semaphores

A widely used synchronization technique is *locking*: when a kernel control path must access a shared data structure or enter a critical region, it must 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 will succeed 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.

Linux offers two kinds of locking: *kernel semaphores*, which are widely used both on uniprocessor systems and multiprocessor ones, and *spin locks*, which are used only on multiprocessors systems. We'll discuss just kernel semaphores here; the other solution will be discussed in the Section 11.4.2 later in this chapter. When a kernel control path tries to acquire a busy resource protected by a kernel semaphore, the corresponding process is suspended. It will become runnable again when the resource is released.

Kernel semaphores are objects of type `struct semaphore` and have these fields:

`count`

Stores an integer value. If it is greater than 0, the resource is free, that is, it is currently available. Conversely, if `count` is less than or equal to 0, the semaphore is busy, that is, the protected resource is currently unavailable. In the latter case, the absolute value of `count` denotes the number of kernel control paths waiting for the resource. Zero means that a kernel control path is using the resource but no other kernel control path 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.

`waking`

Ensures that, when the resource is freed and the sleeping processes is woken up, only one of them succeeds in acquiring the resource. We'll see this field in operation soon.

The `count` field is decremented when a process tries to acquire the lock and incremented when a process releases it. The `MUTEX` and `MUTEX_LOCKED` macros may be used to initialize a semaphore for exclusive access: they set the `count` field, respectively, to 1 (free resource with exclusive access) and (busy resource with exclusive access currently granted to the process that initializes the semaphore). Note that a semaphore could also be initialized with an arbitrary positive value *n* for `count`: in this case, at most *n* processes will be allowed to concurrently access the resource.

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:

```
void down(struct semaphore * sem)
{
    /* BEGIN CRITICAL SECTION */
    --sem->count;
    if (sem->count < 0) {
    /* END CRITICAL SECTION */
        struct wait_queue wait = { current, NULL };
        current->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue(&sem->wait, &wait);
        for (;;) {
            unsigned long flags;
            spin_lock_irqsave(&semaphore_wake_lock, flags);
            if (sem->waking > 0) {
                sem->waking--;
                break;
            }
            spin_unlock_irqrestore(&semaphore_wake_lock, flags);
            schedule(  );
            current->state = TASK_UNINTERRUPTIBLE;
        }
        spin_unlock_irqrestore(&semaphore_wake_lock, flags);
        current->state = TASK_RUNNING;
        remove_wait_queue(&sem->wait, &wait);
    }
}
```

The function decrements the `count` field of the `*sem` semaphore, then checks whether its value is negative. The decrement and the test must be atomically executed, otherwise another kernel control path could concurrently access the field value, with disastrous results (see Section 1.6.5 in Chapter 1). Therefore, these two operations are implemented by means of the following assembly language instructions:

```
movl sem, %ecx
lock /* only for multiprocessor systems */
decl (%ecx)
js 2f
```

On a multiprocessor system, the `decl` instruction is prefixed by a `lock` prefix to ensure the atomicity of the decrement operation (see Section 11.2.2).

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. It is inserted into the wait queue list of the semaphore and put to sleep by directly invoking the `schedule( )` function.

The process is woken up when the resource is freed. Nonetheless, it cannot assume that the resource is now available, since several processes in the semaphore wait queue could be waiting for it. In order to select a winning process, the `waking` field is used: when the releasing process is going to wake up the processes in the wait queue, it increments `waking`; each awakened process then enters a critical region of the `down( )` function and tests whether `waking` is positive. If an awakened process finds the field to be positive, it decrements `waking` and acquires the resource; otherwise it goes back to sleep. The critical region is protected by the `semaphore_wake_lock` global spin lock and by interrupt disabling.

Notice that an interrupt handler or a bottom half must not invoke `down( )`, since this function suspends the process when the semaphore is busy.[2] For that 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.

[2] Exception handlers can block on a semaphore. Linux takes special care to avoid the particular kind of race condition in which two nested kernel control paths compete for the same semaphore; naturally, one of them waits forever because the other cannot run and free the semaphore.

A slightly different function called `down_interruptible( )` is also defined. It is widely used by device drivers since it allows processes that receive a signal while being blocked on a semaphore to give up the "down" operation. If the sleeping process is awakened by a signal before getting the needed resource, the function increments 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`.

When a process releases a kernel semaphore lock, it invokes the `up( )` function, which is essentially equivalent to the following:

```
void up(struct semaphore * sem)
{
    /* BEGIN CRITICAL SECTION */
    ++sem->count;
    if (sem->count <= 0) {
    /* END CRITICAL SECTION */
        unsigned long flags;
        spin_lock_irqsave(&semaphore_wake_lock, flags);
        if (atomic_read(&sem->count) <= 0)
            sem->waking++;
        spin_unlock_irqrestore(&semaphore_wake_lock, flags);
        wake_up(&sem->wait);
    }
}
```

The function increments the `count` field of the `*sem` semaphore, then checks whether its value is negative or null. The increment and the test must be atomically executed, so these two operations are implemented by means of the following assembly language instructions:

```
movl sem, %ecx
lock
incl (%ecx)
jle 2f
```

If the new value of `count` is positive, no process is waiting for the resource, and thus the function terminates. Otherwise, it must wake up the processes in the semaphore wait queue. In order to do this, it increments the `waking` field, which is protected by the `semaphore_wake_lock` spin lock and by interrupt disabling, then invokes `wake_up( )` on the semaphore wait queue.

The increment of the `waking` field is included in a critical region because there can be several processes that concurrently access the same protected resource; therefore, a process could start executing `up( )` while the waiting processes have already been woken up and one of them is already accessing the `waking` field. This also explains why `up( )` checks whether `count` is nonpositive right before incrementing `waking`: another process could have executed the `up( )` function after the first `count` check and before entering the critical region.

We now examine how semaphores are used in Linux. Since the kernel is nonpreemptive, only a few semaphores are needed. Indeed, on a uniprocessor system race conditions usually occur either when a process is blocked during an I/O disk operation or when an interrupt handler accesses a global kernel data structure. Other kinds of race conditions may occur in multiprocessor systems, but in such cases Linux tends to make use of spin locks (see Section 11.4.2 later in this chapter).

The following sections discuss a few typical examples of semaphore use.

11.2.4.1 Slab cache list semaphore

The list of slab cache descriptors (see Section 6.2.2 in Chapter 6) 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. Since the kernel is nonpreemptive, this semaphore plays an active role only in multiprocessor systems.

11.2.4.2 Memory descriptor semaphore

Each memory descriptor of type `mm_struct` includes its own semaphore in the `mmap_sem` field (see Section 7.2 in Chapter 7). 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 us suppose that the kernel must create or extend a memory region for some process; in order 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, any 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.

11.2.4.3 Inode semaphore

This example refers to filesystem handling, which this book has not examined yet. Therefore, we shall limit ourselves to giving the general picture without going into too many details. As we shall see 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, since all processes may (potentially) access the file content, change its name or location, destroy or duplicate it, and so on.

For example, let us suppose that a process is listing 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.

11.2.5 Avoiding Deadlocks on Semaphores

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. A typical deadlock condition occurs when a kernel control path gets the lock for semaphore A and is waiting for semaphore B, while another kernel control path holds the lock for semaphore B and is waiting for semaphore A. Linux has few problems with deadlocks on semaphore requests, since each kernel control path usually needs to acquire just one semaphore at a time.

However, in a couple of cases the kernel must get two semaphore locks. This occurs in the service routines of the `rmdir( )` and the `rename( )` system calls (notice that in both cases two inodes are involved in the operation). In order to avoid such deadlocks, semaphore requests are performed in the order given by addresses: the semaphore request whose `semaphore` data structure is located at the lowest address is issued first.

11.3 The SMP Architecture

Symmetrical multiprocessing (*SMP*) denotes a multiprocessor architecture in which no CPU is selected as the Master CPU, but rather all of them cooperate on an equal basis, hence the name "symmetrical." As usual, we shall focus on Intel SMP architectures.

How many independent CPUs are most profitably included in a multiprocessor system is a hot issue. The troubles are mainly due to the impressive progress reached in the area of cache systems. Many of the benefits introduced by hardware caches are lost by wasting bus cycles in synchronizing the local hardware caches located on the CPU chips. The higher the number of CPUs, the worse the problem becomes.

From the kernel design point of view, however, we can completely ignore this issue: an SMP kernel remains the same no matter how many CPUs are involved. The big jump in complexity occurs when moving from one CPU (a uniprocessor system) to two.

Before proceeding in describing the changes that had to be made to Linux in order to make it a true SMP kernel, we shall briefly review the hardware features of the Pentium dual-processing systems. These features lie in the following areas of computer architecture:

- Shared memory
- Hardware cache synchronization
- Atomic operations
- Distributed interrupt handling
- Interrupt signals for CPU synchronization

Some hardware issues are completely resolved within the hardware, so we don't have to say much about them.

11.3.1 Common Memory

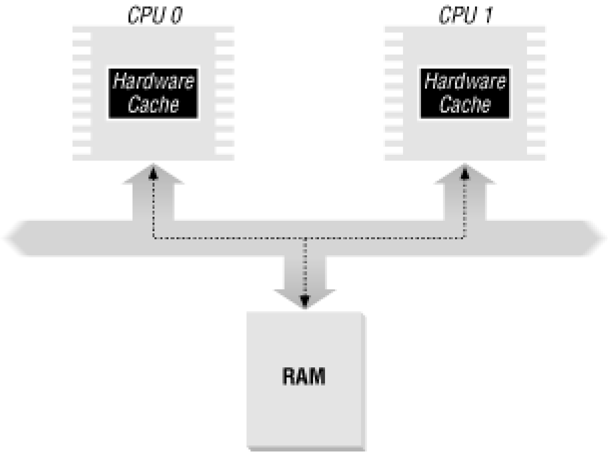
All the CPUs share the same memory; that is, they are connected to a common bus. This means that RAM chips may be accessed concurrently by independent CPUs. Since read or write operations on a RAM chip must be performed serially, a hardware circuit called a *memory arbiter* is inserted between the bus 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. Even uniprocessor systems make use of memory arbiters, since they include a specialized processor called DMA that operates concurrently with the CPU (see Section 13.1.4, in Chapter 13).

In the case of multiprocessor systems, the structure of the arbiter is more complex since 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 bus. From the programming point of view, the arbiter is hidden since it is managed by hardware circuits.

11.3.2 Hardware Support to Cache Synchronization

The section Section 2.4.6 in Chapter 2,explained that the contents of the hardware cache and the RAM maintain their consistency at the hardware level. The same approach holds in the case of a dual processor. As shown in Figure 11-1, 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 and, if so, 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 11-1. The caches in a dual processor



11.3.3 SMP Atomic Operations

Atomic operations for uniprocessor systems have already been introduced in Section 11.2.2. Since standard read-modify-write instructions actually access the memory bus twice, they are not atomic on a multiprocessor system.

Let us give a simple example of what might happen if an SMP kernel used standard instructions. Consider the semaphore implementation described in Section 11.2.4 earlier in this chapter and assume that the `down( )` function decrements and tests the `count` field of the semaphore with a simple `decl` assembly language instruction. What happens if two processes running on two different CPUs simultaneously execute the `decl` instruction on the same semaphore? Well, `decl` is a read-modify-write instruction that accesses the same memory location twice: once to read the old value and again to write the new value.

At first, both CPUs are trying to read the same memory location, but the memory arbiter steps in to grant access to one of them and delay the other. However, when the first read operation is complete the delayed CPU reads exactly the same (old) value from the memory location. Both CPUs then try to write the same (new) value on the memory location; again, the bus memory access is serialized by the memory arbiter, but eventually both write operations will succeed and the memory location will contain the old value decremented by 1. But of course, the global result is completely incorrect. For instance, if `count` was previously set to 1, both kernel control paths will simultaneously gain mutual exclusive access to the protected resource.

Since the early days of the Intel 80286, `lock` instruction prefixes have been introduced to solve that kind of problem. From the programmer's point of view, `lock` is just a special byte that is prefixed to an assembly language instruction. When the control unit detects a `lock` byte, it locks the memory bus so that no other processor can access the memory location specified by the destination operand of the following assembly language instruction. The bus lock is released only when the instruction has been executed. Therefore, read-modify-write instructions prefixed by `lock` are atomic even in a multiprocessor environment.

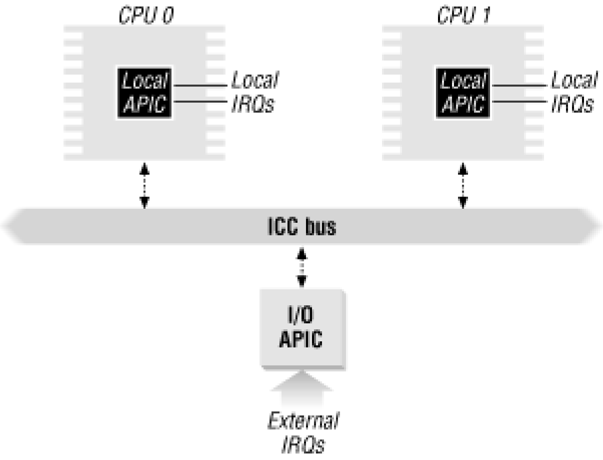
The Pentium allows a `lock` prefix on 18 different instructions. Moreover, some kind of instructions like `xchg` do not require the `lock` prefix because the bus lock is implicitly enforced by the CPU's control unit.

11.3.4 Distributed Interrupt Handling

Being able to deliver interrupts to any CPU in the system is crucial for fully exploiting the parallelism of the SMP architecture. For that reason, Intel has introduced a new component designated as the *I/O APIC* (*I/O Advanced Programmable Interrupt Controller*), which replaces the old 8259A Programmable Interrupt Controller.

Figure 11-2 illustrates in a schematic way the structure of a multi-APIC system. Each CPU chip has its own integrated *Local APIC*. An *Interrupt Controller Communication* (*ICC*) bus connects a 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.

Figure 11-2. APIC system



Each Local APIC has 32-bit registers, an internal clock, a timer device, 240 different interrupt vectors, and two additional IRQ lines reserved for local interrupts, which are typically used to reset the system.

The I/O APIC consists of a set of IRQ lines, a 24-entry *Interrupt Redirection Table*, programmable registers, and a message unit for sending and receiving APIC messages over the ICC 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 any external IRQ signal into a message to one or more Local APIC units via the ICC bus.

Interrupt requests can be distributed among the available CPUs in two ways:

Fixed mode

The IRQ signal is delivered to the Local APICs listed in the corresponding Redirection Table entry.

Lowest-priority mode

The IRQ signal is delivered to the Local APIC of the processor which is executing the process with the lowest priority. Any Local APIC has a programmable *task priority* *register*, which contains the priority of the currently running process. It must be modified by the kernel at each task switch.

Another important feature of the APIC 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 of its own Local APIC. A message is then sent via the ICC bus to the target's Local APIC, which therefore issues a corresponding interrupt to its own CPU.

We'll discuss in Section 11.4.7 later in this chapter how the SMP version of Linux makes use of these interprocessor interrupts.

11.4 The Linux/SMP Kernel

Linux 2.2 support for SMP is compliant with Version 1.4 of the Intel MultiProcessor Specification, which establishes a multiprocessor platform interface standard while maintaining full PC/AT binary compatibility.

As we have seen in Section 11.2.1 earlier in this chapter, race conditions are relatively limited in Linux on a uniprocessor system, so interrupt disabling and kernel semaphores can be used to protect data structures that are asynchronously accessed by interrupt or exception handlers. In a multiprocessor system, however, things are much more complicated: several processes may be running in Kernel Mode, and therefore data structure corruption can occur even if no running process is preempted. The usual way to synchronize access to SMP kernel data structures is by means of semaphores and spin locks (see Section 11.4.2).

Before discussing in detail how Linux 2.2 serializes the accesses to kernel data structures in multiprocessor systems, let us make a brief digression to how this goal was achieved when Linux first introduced SMP support. In order to facilitate the transition from a uniprocessor kernel to a multiprocessor one, the old 2.0 version of Linux/SMP adopted this drastic rule:

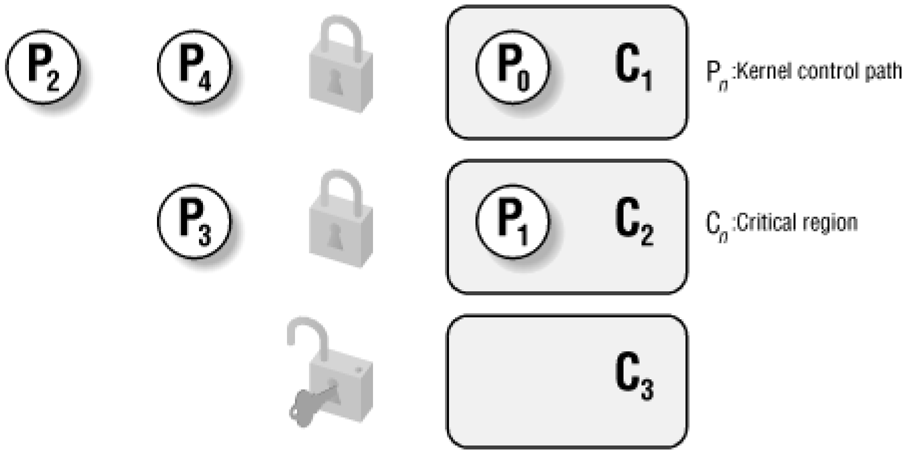
At any given instant, at most one processor is allowed to access the kernel data structures and to handle the interrupts.

This rule dictates that each processor wishing to access the kernel data structures must get a global lock. As long as it holds the lock, it has exclusive access to all kernel data structures. Of course, since the processor will also handle any incoming interrupts, the data structures that are asynchronously accessed by interrupt and exception handlers must still be protected with interrupt disabling and kernel semaphores.

Although very simple, this approach has a serious drawback: processes spend a significant fraction of their computing time in Kernel Mode, therefore this rule may force I/O-bound processes to be sequentially executed. The situation was far from satisfactory, hence the rule was not strictly enforced in the next stable version of Linux/SMP (2.2). Instead, many locks were added, each of which grants exclusive access to single kernel data structure or a single critical region. Therefore, several processes are allowed to concurrently run in Kernel Mode as long as each of them accesses different data structures protected by locks. However, a global kernel lock is still present (see Section 11.4.6 later in this chapter), since not all kernel data structures have been protected with specific locks.

Figure 11-3 illustrates the more flexible Linux 2.2 system. Five kernel control paths—P0, P1, P2, P3, 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 since no kernel control path needs to enter it.

Figure 11-3. Protecting critical regions with several locks



11.4.1 Main SMP Data Structures

In order to handle several CPUs, the kernel must be able to represent the activity that takes place on each of them. In this section we'll consider some significant kernel data structures that have been added to allow multiprocessing.

The most important information is what process is currently running on each CPU, but this information actually does not require a new CPU-specific data structure. Instead, each CPU retrieves the current process through the same `current` macro defined for uniprocessor systems: since it extracts the process descriptor address from the `esp` stack pointer register, it yields a value that is CPU-dependent.

A first group of new CPU-specific variables refers to the SMP architecture. Linux/SMP has a hard-wired limit on the number of CPUs, which is defined by the `NR_CPUS` macro (usually 32).

During the initialization phase, Linux running on the booting CPU probes whether other CPUs exist (some CPU slots of an SMP board may be empty). As a result, both a counter and a bitmap are initialized: `max_cpus` stores the number of existing CPUs while `cpu_present_map` specifies which slots contain a CPU.

An existing CPU is not necessarily activated, that is, initialized and recognized by the kernel. Another pair of variables, a counter called `smp_num_cpus` and a bitmap called `cpu_online_map`, keeps track of the activated CPUs. If some CPU cannot be properly initialized, the kernel clears the corresponding bit in `cpu_online_map`.

Each active CPU is identified in Linux by a sequential logical number called *CPU ID*, which does not necessarily coincide with the CPU slot number. The `cpu_number_map` and `__cpu_logical_map` arrays allow conversion between CPU IDs and CPU slot numbers.

The process descriptor includes the following fields representing the relationships between the process and a processor:

`has_cpu`

Flag denoting whether the process is currently running (value 1) or not running (value 0)

`processor`

Logical number of the CPU that is running the process, or `NO_PROC_ID` if the process is not running

The `smp_processor_id( )` macro returns the value of `current->processor`, that is, the logical number of the CPU that executes the process.

When a new process is created by `fork( )`, the `has_cpu` and `processor` fields of its descriptor are initialized respectively to and to the value `NO_PROC_ID`. When the `schedule( )` function selects a new process to run, it sets its `has_cpu` field to 1 and its `processor` field to the logical number of the CPU that is doing the task switch. The corresponding fields of the process being replaced are set to and to `NO_PROC_ID`, respectively.

During system initialization `smp_num_cpus` different *swapper* processes are created. Each of them has a PID equal to and is bound to a specific CPU. As usual, a *swapper* process is executed only when the corresponding CPU is idle.

11.4.2 Spin Locks

Spin locks are a locking mechanism designed to work in a multiprocessing environment. They are similar to the kernel semaphores described earlier, except that when a process finds the lock closed by another process, it "spins" around repeatedly, executing a tight instruction loop.

Of course, spin locks would be useless in a uniprocessor environment, since the waiting process would keep running, and therefore the process that is holding the lock would not have any chance to release it. In a multiprocessing environment, however, spin locks are much more convenient, since their overhead is very small. In other words, a context switch takes a significant amount of time, so it is more efficient for each process to keep its own CPU and simply spin while waiting for a resource.

Each spin lock is represented by a `spinlock_t` structure consisting of a single `lock` field; the values and 1 correspond, respectively, to the "unlocked" and the "locked" state. The `SPIN_LOCK_UNLOCKED` macro initializes a spin lock to 0.

The functions that operate on spin locks are based on atomic read/modify/write operations; this ensures that the spin lock will be properly updated by a process running on a CPU even if other processes running on different CPUs attempt to modify the spin lock at the same time.[3]

[3] Spin locks, ironically enough, are global and therefore must themselves be protected against concurrent access.

The spin_lock macro is used to acquire a spin lock. It takes the address slp of the spin lock as its parameter and yields essentially the following code:

```
1: lock; btsl $0, slp
   jnc  3f
2: testb $1,slp
   jne 2b
   jmp 1b
3:
```

The btsl atomic instruction copies into the carry flag the value of bit in *slp, then sets the bit. A test is then performed on the carry flag: if it is null, it means that the spin lock was unlocked and hence normal execution continues at label 3 (the f suffix denotes the fact that the label is a "forward" one: it appear 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 the value 0. Then execution restarts from label 1, since it would be unsafe to proceed without checking whether another processor has grabbed the lock.[4]

[4] The actual implementation of spin_lock 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 (free spin lock) the hardware cache is not filled with code that won't be executed. In our discussion we omit these optimization details.

The spin_unlock macro releases a previously acquired spin lock; it essentially yields the following code:

```
lock; btrl $0, slp
```

The btrl atomic assembly language instruction clears the bit of the spin lock *slp.

Several other macros have been introduced to handle spin locks; their definitions on a multiprocessor system are described in Table 11-3 (see Table 11-2 for their definitions on a uniprocessor system).

Table 11-3. Spin Lock Macros on a Multiprocessor System

Macro	Description
spin_lock_init(slp)	Set slp->lock to 0
spin_trylock (slp)	Set slp->lock to 1, return 1 if got the lock, otherwise
spin_unlock_wait(slp)	Cycle until slp->lock becomes 0
spin_lock_irq(slp)	_ _cli( );spin_lock(slp)
spin_unlock_irq(slp)	spin_unlock(slp);__sti( )

spin_lock_irqsave(slp,flags)	__save_flags(flags);__cli( );
	spin_lock(slp)

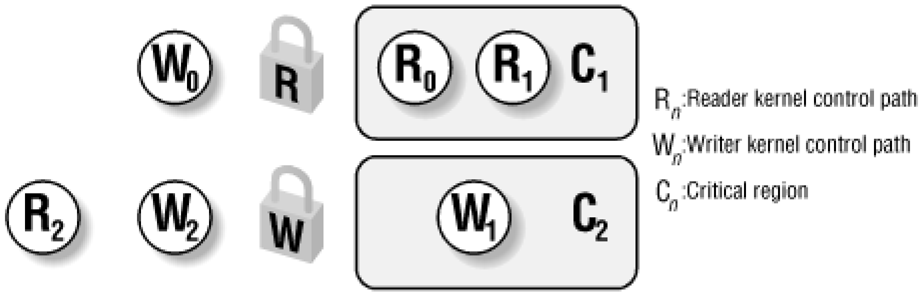
spin_unlock_irqrestore(slp,flags)	spin_unlock(slp);_ _restore_flags(flags)

11.4.3 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 11-4 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 11-4. Read/write spin locks


W_0
R
R_0
R_1
C_1
R_2
W_2
W
W_1
C_2
R_n:Reader kernel control path
W_n:Writer kernel control path
C_n:Critical region

Each read/write spin lock is a `rwlock_t` structure; its `lock` field is a 32-bit counter that represents the number of kernel control paths currently reading the protected data structure. The highest-order bit of the `lock` field is the write lock: it is set when a kernel control path is modifying the data structure.[5] The `RW_LOCK_UNLOCKED` macro initializes the `lock` field of a read/write spin lock to 0. The `read_lock` macro, applied to the address `rwlp` of a read/write spin lock, essentially yields the following code:

```
1: lock; incl rwlp
   jns 3f
   lock; decl rwlp
2: cmpl $0, rwlp
   js 2b
   jmp 1b
3:
```

After increasing by 1 the value of `rwlp->lock`, the function checks whether the field has a negative value—that is, if it is already locked for writing. If not, execution continues at label 3. Otherwise, the macro restores the previous value and spins around until the highest-order bit becomes 0; then it starts back from the beginning.

The `read_unlock` function, applied to the address `rwlp` of a read/write spin lock, yields the following assembly language instruction:

```
lock; decl rwlp
```

The `write_lock` function applied to the address `rwlp` of a read/write spin lock yields the following instructions:

[5] It would also be set if there are more than 2,147,483,647 readers: of course, such a huge limit is never reached.

```
1: lock; btsl $31, rwlp
   jc 2f
   testl $0x7fffffff, rwlp
   je 3f
   lock; btrl $31, rwlp
2: cmp $0, rwlp
   jne 2b
   jmp 1b
3:
```

The highest-order bit of `rwlp->lock` is set. If its old value was 1, the write lock is already busy, and therefore the execution continues at label `2`. Here the macro executes a tight loop waiting for the `lock` field to become (meaning that the write lock was released). If the old value of the highest-order bit was (meaning there is no write lock), the macro checks whether there are readers. If so, the write lock is released and the macro waits until `lock` becomes 0; otherwise, the CPU has the exclusive access to the resource, so execution continues at label `3`.

Finally, the `write_unlock` macro, applied to the address `rwlp` of a read/write spin lock, yields the following instruction:

```
lock; btrl $31, rwlp
```

Table 11-4 lists the interrupt-safe versions of the macros described in this section.

Table 11-4. Read/Write Spin Lock Macros on a Multiprocessor System

Function	Description
`read_lock_irq(rwlp)`	`_ _cli( ); read_lock(rwlp)`
`read_unlock_irq(rwlp)`	`read_unlock(rwlp); _ _sti( )`
`write_lock_irq(rwlp)`	`_ _cli( ); write_lock(rwlp)`
`write_unlock_irq(rwlp)`	`write_lock(rwlp); _ _sti( )`
`read_lock_irqsave(rwlp,flags)`	`__save_flags(flags); __cli( ); read_lock(rwlp)`
`read_unlock_irqrestore(rwlp,flag)`	`read_unlock(rwlp); __restore_flags(flags)`
`write_lock_irqsave(rwlp,flags)`	`__save_flags(flags); __cli( ); write_lock(rwlp)`
`write_unlock_irqrestore(rwlp,flags)`	`write_unlock(rwlp); __restore_flags(flags)`

11.4.4 Linux/SMP Interrupt Handling

We stated previously that, on Linux/SMP, interrupts are broadcast by the I/O APIC to all Local APICs; that is, to all CPUs. This means that all CPUs having the `IF` flags set will receive the same interrupt. However, only one CPU must handle the interrupt, although all of them must acknowledge to their Local APICs they received it.

In order to do this, each IRQ main descriptor (see Section 4.6.2 in Chapter 4) includes an `IRQ_INPROGRESS` flag. If it is set, the corresponding interrupt handler is already running on some CPU. Therefore, when each CPU acknowledges to its Local APIC that the interrupt was accepted, it checks whether the flag is already set. If it is, the CPU does not handle the interrupt and exits back to what it was running; otherwise, the CPU sets the flag and starts executing the interrupt handler.

Of course, accesses to the IRQ main descriptor must be mutually exclusive; therefore, each CPU always acquires the `irq _controller_lock` spin lock before checking the value of `IRQ _INPROGRESS`. The same lock also prevents several CPUs from fiddling with the interrupt controller simultaneously; this precaution is necessary for old SMP machines that have just one external interrupt controller shared by all CPUs.

The `IRQ _INPROGRESS` flag ensures that each specific interrupt handler is atomic with respect to itself among all CPUs. However, several CPUs may concurrently handle different interrupts. The `global_irq _count` variable contains the number of interrupt handlers that are being handled at each given instant on all CPUs. This value could be greater than the number of CPUs, since any interrupt handler can be interrupted by another interrupt handler of a different kind. Similarly, the `local_irq _count` array stores the number of interrupt handlers being handled on each CPU.

As we have already seen, the kernel must often disable interrupts in order to prevent corruption of a kernel data structure that may be accessed by interrupt handlers. Of course, local CPU interrupt disabling provided by the `__cli( )` macro is not enough, since it does not prevent some other CPU from accessing the kernel data structure. The usual solution consists of acquiring a spin lock with an IRQ-safe macro (like `spin_lock_irqsave`).

In a few cases, however, interrupts should be disabled on all CPUs. In order to achieve such a result, the kernel does not clear the `IF` flags on all CPUs; instead it uses the `global_irq _lock` spin lock to delay the execution of the interrupt handlers. The `global_irq _holder` variable contains the logical identifier of the CPU that is holding the lock. The `get_irqlock( )` function acquires the spin lock and waits for the termination of all interrupt handlers running on the other CPUs. Moreover, if the caller is not a bottom half itself, the function waits for the termination of all bottom halves running on the other CPUs. No further interrupt handler on other CPUs will start running until the lock is released by invoking `release_irqlock( )`.

Global interrupt disabling is performed by the `cli( )` macro, which just invokes the `__global_cli( )` function:

```
__save_flags(flags);
if (!(flags & (1 << 9))) /* testing IF flag */
    return;
cpu = smp_processor_id( );
__cli( );
if (!local_irq_count[cpu])
    return;
get_irqlock(cpu);
```

Notice that global interrupt disabling is not performed when the CPU is running with local interrupts already disabled or when the CPU is running an interrupt handler itself.[6]

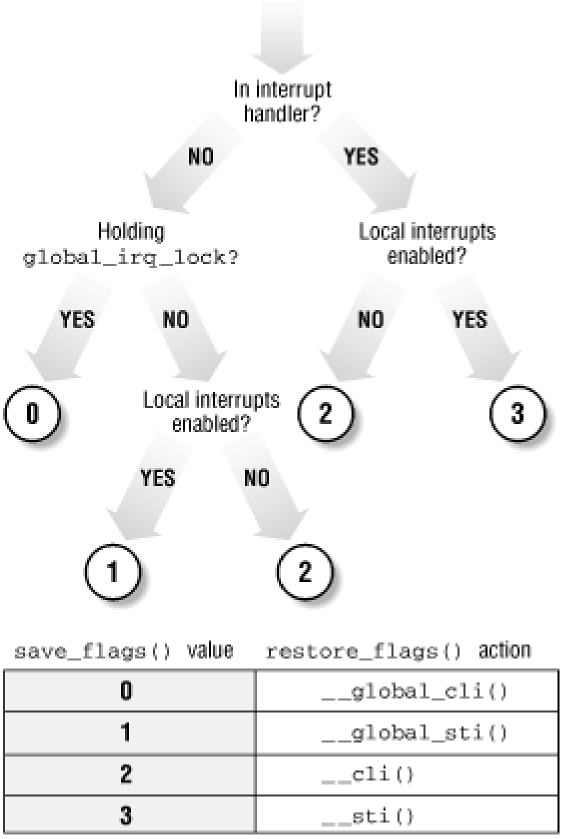
[6] Deadlock conditions can easily occur if such constraints are removed. For instance, suppose that `cli( )` could "promote" a local interrupt disabling to a global one. Consider a kernel control path that is executing a critical region protected by some spin lock and with local interrupt disabled. The critical region can legally include a `cli( )` macro, since it could invoke a function that is also accessed with local interrupts enabled. The `get_irqlock( )` function starts waiting for interrupt handlers to complete on the other CPUs. However, an interrupt handler in another kernel control path could be stuck on the spin lock that protects the critical region, waiting for the first kernel control path to release it: deadlock!

Global interrupt enabling is performed by the `sti( )` macro, which just invokes the `__global_sti( )` function:

```
cpu = smp_processor_id( );
if (!local_irq_count[cpu])
    release_irqlock(cpu);
__sti( );
```

Linux also provides SMP versions of the `_ _save_flags` and `__restore_flags` macros, which are called `save_flags` and `restore_flags`: they save and reload, respectively, information controlling the interrupt handling for the executing CPU. As illustrated in Figure 11-5, `save_flags` yields an integer value that depends on three conditions; `restore_flags` performs actions based on the value yielded by `save_flags`.

Figure 11-5. Actions performed by save_ flags( ) and restore_ flags( )



save_flags() value	restore_flags() action
0	__global_cli()
1	__global_sti()
2	__cli()
3	__sti()

Finally, the `synchronize_irq( )` function is called when a kernel control path wishes to synchronize itself with all interrupt handlers:

```
if (atomic_read(&global_irq_count)) {
    cli();
    sti();
}
```

By invoking `cli( )`, the function acquires the `global_irq _lock` spin lock and then waits until all executing interrupt handlers terminate; once this is done, it reenables interrupts. The `synchronize_irq( )` function is usually called by device drivers when they want to make sure that all activities carried on by interrupt handlers are over.

11.4.5 Linux/SMP Bottom Half Handling

Bottom halves are handled much like interrupt handlers, but no bottom half can ever run concurrently with other bottom halves. Moreover, disabling interrupts also disables the running of bottom halves. The `global_bh_count` variable is a flag that specifies whether a bottom half is currently active on some CPU. The `synchronize_bh( )` function is called when a kernel control path must wait for the termination of a currently executing bottom half.

The `global_bh_lock` variable is used to disable the execution of bottom halves on all CPUs; in other words, it ensures that some critical region is atomic with respect to all bottom halves on all CPUs.

The `start_bh_atomic( )` function, which locks out bottom halves, consists of:

```
atomic_inc(&global_bh_lock);
synchronize_bh();
```

The complementary `end_bh_atomic( )` function is used to reenable the bottom halves by executing:

```
atomic_dec(&global_bh_lock);
```

Therefore, the `do_bottom_half( )` function starts bottom halves only if:

- No other bottom half is currently running on any CPU (`global_bh_count` is null).
- The bottom halves are not disabled (`global_bh_lock` is null).
- No interrupt handler is running on any CPU (`global_irq_count` is null).
- Interrupts are globally enabled (`global_irq_lock` is free).

Serial execution of bottom halves is inherited from previous versions of Linux. Allowing bottom halves to be executed concurrently would require a full revision of all device drivers that use them.

11.4.6 Global and Local Kernel Locks

As we have already mentioned, in the Version 2.2 of Linux/SMP a *global kernel lock* named `kernel_flag` is still widely used. In Version 2.0, this spin lock was relatively crude, ensuring simply that only one processor at a time could run in Kernel Mode. The 2.2 kernel is considerably more flexible and no longer relies on a single spin lock; however, it is still used to protect a very large number of kernel data structures, namely:

- All data structures related to the Virtual Filesystem and to file handling (see Chapter 12)
- Most kernel data structures related to networking
- All kernel data structures for interprocess communication (IPC); see Chapter 18
- Several less important kernel data structures

The global kernel lock still exists because introducing new locks is not trivial: both deadlocks and race conditions must be carefully avoided.

All system call service routines related to files, including the ones related to file memory mapping, must acquire the global kernel lock before starting their operations and must release it when they terminate. Therefore, a very large number of system calls cannot concurrently execute on Linux/SMP.

Every process descriptor includes a `lock_depth` field, which allows the same process to acquire the global kernel lock several times. Therefore, two consecutive requests for it will not hang the processor (as for normal spin locks). If the process does not want the lock, the field has the value -1. If the process wants it, the field value plus 1 specifies how many times the lock has been requested. The `lock_depth` field is crucial for interrupt handlers, exception handlers, and bottom halves. Without it, any asynchronous function that tries to get the global 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 global kernel lock. The former function is equivalent to:

```
if (++current->lock_depth == 0)
    spin_lock(&kernel_flag);
```

while the latter is equivalent to:

```
if (--current->lock_depth < 0)
    spin_unlock(&kernel_flag);
```

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 global kernel lock before terminating.

Although the global kernel lock still protects a large number of kernel data structures, work is in progress to reduce that number by introducing many additional smaller locks. Table 11-5 lists some kernel data structures that are already protected by specific (read/write) spin locks.

Table 11-5. Various Kernel Spin Locks

Spin Lock	Protected Resource
`console_lock`	Console
`dma_spin_lock`	DMA's data structures
`inode_lock`	Inode's data structures
`io_request_lock`	Block IO subsystem
`kbd_controller_lock`	Keyboard
`page_alloc_lock`	Buddy system's data structures
`runqueue_lock`	Runqueue list
`semaphore_wake_lock`	Semaphores's waking fields
`tasklist_lock (rw)`	Process list
`taskslot_lock`	List of free entries in task
`timerlist_lock`	Dynamic timer lists
`tqueue_lock`	Task queues' lists
`uidhash_lock`	UID hash table
`waitqueue_lock (rw)`	Wait queues' lists
`xtime_lock (rw)`	`xtime` and `lost_ticks`

As already explained, finer granularity in the lock mechanism enhances system performance, since less serialization is enforced among the processors. For instance, a kernel control path that accesses the runqueue list is allowed to concurrently run with another kernel control path that is servicing a file-related system call. Similarly, using a read/write lock, two kernel control paths may concurrently access the process list as long as neither of them wants to modify it.

11.4.7 Interprocessor Interrupts

Interprocessor interrupts (in short, IPIs) are part of the SMP architecture and are actively used by Linux in order to exchange messages among CPUs. Linux/SMP provides the following functions to handle them:

`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_single( )`

Sends an IPI to a single, specified CPU

Depending on the I/O APIC configuration, the kernel may sometimes need to invoke the `send_IPI_self( )` function. The other functions are used to implement interprocessor messages.

Linux/SMP recognizes five kinds of messages, which are interpreted by the receiving CPU as different interrupt vectors:

`RESCHEDULE_VECTOR` (`0x30`)

Sent to a single CPU in order to force the execution of the `schedule( )` function on it. The corresponding Interrupt Service Routine (ISR) is named `smp_reschedule_interrupt( )`. This message is used by `reschedule_idle( )` and by `send_sig_info( )` to preempt the running process on a CPU.

`INVALIDATE_TLB_VECTOR` (`0x31`)

Sent to all CPUs but the sender, forcing them to invalidate their translation lookaside buffers. The corresponding ISR, named `smp_invalidate_interrupt( )`, invokes the `_ _flush_tlb( )` function.[7] This message is used whenever the kernel modifies a page table of some process.

[7] A subtle concurrency problem occurs when trying to flush the translation lookaside buffers of all processors while some of them run with the interrupts disabled. Therefore, while spinning in tight loops, the kernel control paths keep checking whether some CPU has sent an "invalidate TLB" message.

`STOP_CPU_VECTOR` (`0x40`)

Sent to all CPUs but the sender, forcing the receiving CPUs to halt. The corresponding Interrupt Service Routine is named `smp_stop_cpu_interrupt( )`. This message is used only when the kernel detects an unrecoverable internal error.

`LOCAL_TIMER_VECTOR` (`0x41`)

A timer interrupt automatically sent to all CPUs by the I/O APIC. The corresponding Interrupt Service Routine is named `smp_apic_timer_interrupt( )`.

`CALL_FUNCTION_VECTOR` (`0x50`)

Sent to all CPUs but the sender, forcing those CPUs to run a function passed by the sender. The corresponding ISR is named `smp_call_function_interrupt( )`. A typical use of this message is to force CPUs to synchronize and to reload the state of the Memory Type Range Registers (MTRRs). Starting with the Pentium Pro model, Intel microprocessors include these additional registers to easily customize cache operations. Linux uses 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.

11.5 Anticipating Linux 2.4

Linux 2.4 changes a bit the way semaphores are implemented. Essentially, they are now more efficient because, when a semaphore is released, usually only one sleeping process is awoken.

As already mentioned, Linux 2.4 enhances support for high-end SMP architectures. It is now possible to make use of multiple external I/O APIC chips, and all the code that handles interprocessor interrupts (IPIs) has been rewritten.

However, the most important change is that Linux 2.4 is much more multithreaded than Linux 2.2. In other words, it makes use of many new spin locks and reduces the role of the global kernel lock, particularly in the networking code. Linux 2.4 is therefore much more efficient on SMP architectures and performs much better as a high-end server.

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 disk types in the same way other Unix variants do, through a concept called the Virtual Filesystem.

The idea behind the Virtual Filesystem is that the internal objects representing files and filesystems in kernel memory embody a wide range of information; there is a field or function to support any operation provided by any real filesystem supported by Linux. For each read, write, or other function called, the kernel substitutes the actual function that supports a native Linux filesystem, the NT filesystem, or whatever other filesystem the file is on.

This chapter discusses the aims, the structure, and the 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 will be covered in Chapter 13, while pipes will be discussed in Chapter 18. To show how a real filesystem works, Chapter 17, covers the Second Extended Filesystem that appears on nearly all Linux systems.

12.1 The Role of the VFS

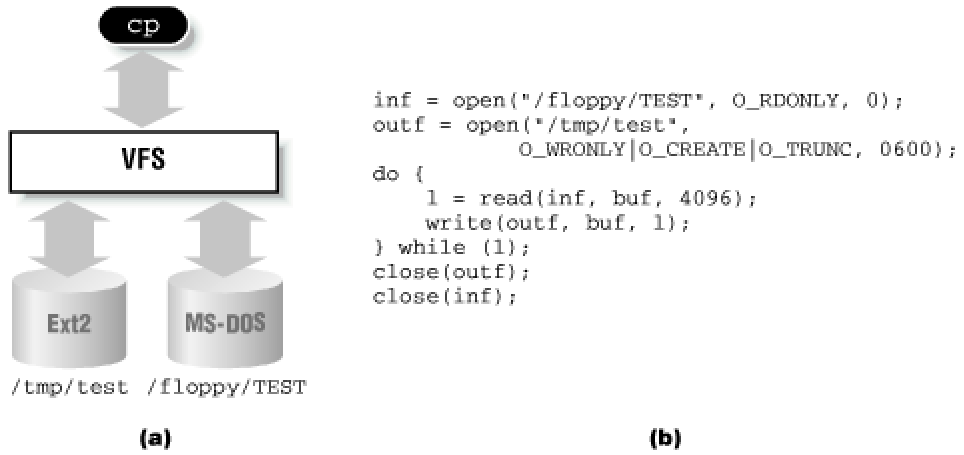
The *Virtual Filesystem* (also known as Virtual Filesystem Switch or VFS) is a kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.

For instance, let us 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 Ext2 (Second Extended Filesystem) directory. As shown in Figure 12-1 (a), the VFS is an abstraction layer between the application program and the filesystem implementations. 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 well known to anyone who has done Unix programming (see also Section 1.5.6 in Chapter 1); the code executed by *cp* is shown in Figure 12-1 (b).

Figure 12-1. VFS role in a simple file copy operation



```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
           O_WRONLY|O_CREATE|O_TRUNC, 0600);
do {
    l = read(inf, buf, 4096);
    write(outf, buf, l);
} while (l);
close(outf);
close(inf);
```

Filesystems supported by the VFS may be grouped into three main classes:

Disk-based filesystems

Manage the memory space available in a local disk partition. The official Linux disk-based filesystem is Ext2. Other well-known disk-based filesystems supported by the VFS are:

- Filesystems for Unix variants like System V and BSD
- Microsoft filesystems like MS-DOS, VFAT (Windows 98), and NTFS (Windows NT)
- ISO9660 CD-ROM filesystem (formerly High Sierra Filesystem)
- Other proprietary filesystems like HPFS (IBM's OS/2), HFS (Apple's Macintosh), FFS (Amiga's Fast Filesystem), and ADFS (Acorn's machines)

Network filesystems

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's filesystem), SMB (Microsoft's Windows and IBM's OS/2 LAN Manager), and NCP (Novell's NetWare Core Protocol).

Special filesystems (also called *virtual filesystems*)

Do not manage disk space. Linux's `/proc` filesystem provides a simple interface that allows users to access the contents of some kernel data structures. The `/dev/pts` filesystem is used for pseudo-terminal support as described in the Open Group's Unix98 standard.

In this book we describe only the Ext2 filesystem, which is the topic of Chapter 17; the other filesystems will not be covered for lack of space.

As mentioned in Section 1.5 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. All other filesystems can be "mounted" on subdirectories of the root filesystem.[1]

[1] When a filesystem is mounted on some directory, the contents of the directory in the parent filesystem are no longer accessible, since any pathname including the mount point will refer to the mounted filesystem. However, the original directory's content will show 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.

A disk-based filesystem is usually stored in a hardware block device like a hard disk, a floppy, or a CD-ROM. A useful feature of Linux's VFS allows it to handle *virtual block devices* like */dev/loop0*, which may be used to mount filesystems stored in regular files. As a possible application, a user may protect his 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.

12.1.1 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, since 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.

For instance, in the common file model each directory is regarded as a normal file, which contains a list of files and other directories. However, several non-Unix disk-based filesystems make use of a File Allocation Table (FAT), which stores the position of each file in the directory tree: in these filesystems, directories are not files. In order 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 `sys_read( )`, just like any other system call. The file is represented by a `file` data structure in kernel memory, as we shall see later in the 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, 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 like C++. Objects are thus implemented as 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*

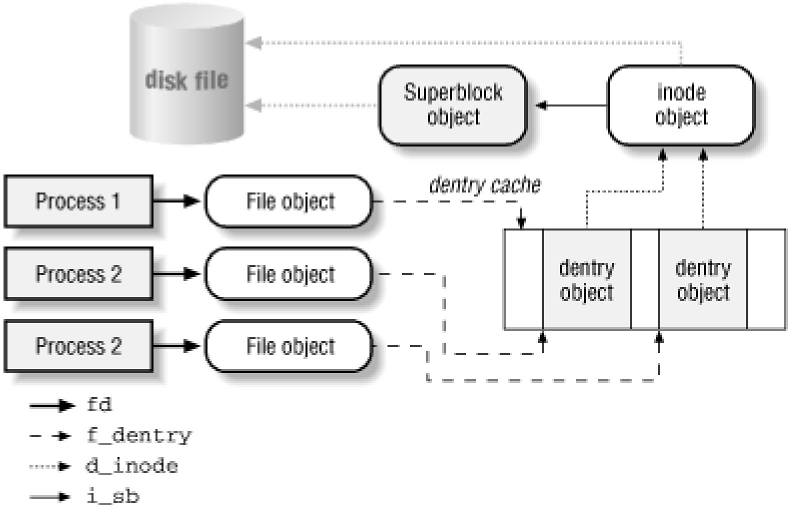
Stores information about the interaction between an open file and a process. This information exists only in kernel memory during the period each process accesses a file.

The *dentry object*

Stores information about the linking of a directory entry 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 makes use of 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.[2] Beside the dentry cache, Linux uses other disk caches, like the buffer cache and the page cache, which will be described in forthcoming chapters.

[2] 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 Section 2.4.6 in Chapter 2). A memory cache is a software mechanism introduced to bypass the Kernel Memory Allocator (see Section 6.2.1 in Chapter 6).

12.1.2 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 and hence will be discussed in later chapters. A last group of system calls handled by the VFS, such as `socket( )`, `connect( )`, `bind( )`, and `protocols( )`, refer to sockets and are used to implement networking; they will not be covered in this book. 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 17.

Table 12-1. Some System Calls Handled by the VFS

System Call Name	Description
`mount( ) umount( )`	Mount/Unmount filesystems
`sysfs( )`	Get filesystem information
`statfs( ) fstatfs( ) ustat( )`	Get filesystem statistics
`chroot( )`	Change root directory
`chdir( ) fchdir( ) getcwd( )`	Manipulate current directory
`mkdir( ) rmdir( )`	Create and destroy directories
`getdents( ) readdir( ) link( ) unlink( ) rename( )`	Manipulate directory entries
`readlink( ) symlink( )`	Manipulate soft links
`chown( ) fchown( ) lchown( )`	Modify file owner
`chmod( ) fchmod( ) utime( )`	Modify file attributes
`stat( ) fstat( ) lstat( ) access( )`	Read file status
`open( ) close( ) creat( ) umask( )`	Open and close files
`dup( ) dup2( ) fcntl( )`	Manipulate file descriptors
`select( ) poll( )`	Asynchronous I/O notification
`truncate( ) ftruncate( )`	Change file size
`lseek( ) _llseek( )`	Change file pointer
`read( ) write( ) readv( ) writev( ) sendfile( )`	File I/O operations
`pread( ) pwrite( )`	Seek file and access it
`mmap( ) munmap( )`	File memory mapping
`fdatasync( ) fsync( ) sync( ) msync( )`	Synchronize file data
`flock( )`	Manipulate file lock

We said earlier that the VFS is a layer between application programs and specific filesystems. 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 does not have to invoke a specific filesystem procedure. In some sense, the VFS could be considered as a "generic" filesystem that relies, when necessary, on specific ones.

12.2 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.

12.2.1 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
kdev_t	s_dev	Device identifier
unsigned long	s_blocksize	Block size in bytes
unsigned char	s_blocksize_bits	Block size in number of bits
unsigned char	s_lock	Lock flag
unsigned char	s_rd_only	Read-only flag
unsigned char	s_dirt	Modified (dirty) flag
struct file_system_type *	s_type	Filesystem type
struct super_operations *	s_op	Superblock methods
struct dquot_operations *	dq_op	Disk quota methods
unsigned long	s_flags	Mount flags
unsigned long	s_magic	Filesystem magic number
unsigned long	s_time	Time of last superblock change
struct dentry *	s_root	Dentry object of mount directory
struct wait_queue *	s_wait	Mount wait queue
struct inode *	s_ibasket	Future development
short int	s_ibasket_count	Future development
short int	s_ibasket_max	Future development
struct list_head	s_dirty	List of modified inodes
union	u	Specific filesystem information

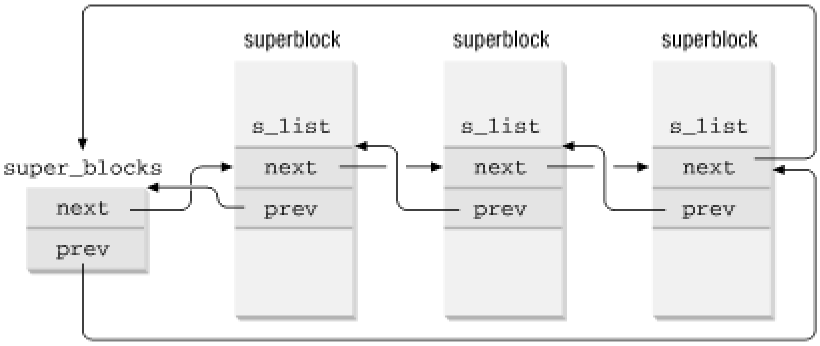
All superblock objects (one per mounted filesystem) are linked together in a circular doubly linked list. The addresses of the first and last elements of the list are stored in the next and prev fields, respectively, of the s_list field in the super_blocks variable. This field has the data type struct list_head, which is also found in the s_dirty field of the superblock and in a number of other places in the kernel; it consists simply of pointers to the next and previous elements of a list. Thus, the s_list field of a superblock object includes the pointers to the two adjacent superblock objects in the list. Figure 12-3 illustrates how the list_head elements, next and prev, are embedded in the superblock object.

The last u union field includes superblock information that belongs to a specific filesystem; for instance, as we shall see later in Chapter 17, if the superblock object refers to an Ext2 filesystem, the field stores 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 in the u field is duplicated in memory for reasons of efficiency. Any 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 u union 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 shall see in Section 14.1.5 in Chapter 14, Linux minimizes this problem by periodically copying all dirty superblocks to disk.

Figure 12-3. The superblock list



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:

```
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 thus directly invoked.

Let us 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:

`read_inode(inode)`

Fills the fields of the inode object whose address is passed as the parameter from the data on disk; the `i_ino` field of the inode object identifies the specific filesystem inode on disk to be read.

`write_inode(inode)`

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.

`put_inode(inode)`

Releases the inode object whose address is passed as the parameter. As usual, releasing an object does not necessarily mean freeing memory since other processes may still use that object.

`delete_inode(inode)`

Deletes the data blocks containing the file, the disk inode, and the VFS inode.

`notify_change(dentry, iattr)`

Changes some attributes of the inode according to the `iattr` parameter. If the `notify_change` field is `NULL`, the VFS falls back on the `write_inode( )` method.

`put_super(super)`

Releases the superblock object whose address is passed as the parameter (because the corresponding filesystem is unmounted).

`write_super(super)`

Updates a filesystem superblock with the contents of the object indicated.

`statfs(super, buf, bufsize)`

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)`

Like `put_inode`, but also releases all pages that contain data concerning the file that corresponds to the indicated inode.

`umount_begin(super)`

Interrupts a mount operation, because the corresponding unmount operation has been started (used only by network filesystems).

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 `read_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 the `read_super` method in another object describing the filesystem type (seelater Section 12.3).

12.2.2 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 list_head`	`i_hash`	Pointers for the hash list
`struct list_head`	`i_list`	Pointers for the inode list
`struct list_head`	`i_dentry`	Pointers for the dentry list
`unsigned long`	`i_ino`	inode number
`unsigned int`	`i_count`	Usage counter
`kdev_t`	`i_dev`	Device identifier
`umode_t`	`i_mode`	File type and access rights
`nlink_t`	`i_nlink`	Number of hard links
`uid_t`	`i_uid`	Owner identifier
`gid_t`	`i_gid`	Group identifier
`kdev_t`	`i_rdev`	Real device identifier
`off_t`	`i_size`	File length in bytes
`time_t`	`i_atime`	Time of last file access
`time_t`	`i_mtime`	Time of last file write
`time_t`	`i_ctime`	Time of last inode change
`unsigned long`	`i_blksize`	Block size in bytes
`unsigned long`	`i_blocks`	Number of blocks of the file
`unsigned long`	`i_version`	Version number, automatically incremented after each use
`unsigned long`	`i_nrpages`	Number of pages containing file data
`struct semaphore`	`i_sem`	inode semaphore
`struct semaphore`	`i_atomic_write`	inode semaphore for atomic write
`struct inode_operations *`	`i_op`	inode operations
`struct super_block *`	`i_sb`	Pointer to superblock object
`struct wait_queue *`	`i_wait`	inode wait queue
`struct file_lock *`	`i_flock`	Pointer to file lock list
`struct vm_area_struct *`	`i_mmap`	Pointer to memory regions used to map the file
`struct page *`	`i_pages`	Pointer to page descriptor
`struct dquot **`	`i_dquot`	inode disk quotas
`unsigned long`	`i_state`	inode state flag
`unsigned int`	`i_flags`	Filesystem mount flag
`unsigned char`	`i_pipe`	True if file is a pipe
`unsigned char`	`i_sock`	True if file is a socket
`int`	`i_writecount`	Usage counter for writing process
`unsigned int`	`i_attr_flags`	File creation flags
`_ _u32`	`i_generation`	Reserved for future development
`union`	`u`	Specific filesystem information

The final `u` union field is used to include inode information that belongs to a specific filesystem. For instance, as we shall see in Chapter 17, if the inode object refers to an Ext2 file, the field stores an `ext2_inode_info` structure.

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`, the inode is dirty, that is, the corresponding disk inode must be updated. Other values of the `i_state` field are `I_LOCK` (which means that the inode object is locked) and `I_FREEING` (which means that the inode object is being freed).

Each inode object always appears in one of the following circular doubly linked lists:

- The list of unused inodes. 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 memory cache.
- The list of in-use inodes. 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 together the `i_list` fields of the proper inode objects.

Inode objects belonging to the "in use" or "dirty" lists 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 filesystem that includes the file.[3] Since 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.

[3] Actually, a Unix process may open a file and then unlink it: the `i_nlink`field of the inode could become 0, yet the process is still able to act on the file. In this particular case, the inode is removed from the hash table, even if it still belongs to the in-use or dirty list.

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. The structure also includes a pointer to the file operation methods (see Section 12.2.3). Here are the inode operations, in the order they appear in the `inode_operations` table:

`create(dir, dentry, mode)`

Creates a new disk inode for a regular file associated with a dentry object in some directory.

`lookup(dir, dentry)`

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 number.

`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 memory area specified by `buffer` the file pathname corresponding to the symbolic link specified by the dentry.

`follow_link(inode, dir)`

Translates a symbolic link specified by an inode object; if the symbolic link is a relative pathname, the lookup operation starts from the specified directory.

`readpage(file, pg)`

Reads a page of data from an open file. As we shall see in Chapter 15, regular files are read by this method.

`writepage(file, pg)`

Writes a page of data into an open file. Most filesystems do not make use of this method when writing regular files.

`bmap(inode, block)`

Returns the logical block number corresponding to the file block number of the file associated with an inode.

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)

Checks whether the specified access mode is allowed for the file associated with inode.

smap(inode, sector)

Similar to bmap( ), but determines the disk sector number; used by FAT-based filesystems.

updatepage(inode, pg, buf, offset, count, sync)

Updates, if needed, a page of data of a file associated with an inode (usually invoked by network filesystems, which may have to wait a long time before updating remote files).

revalidate(dentry)

Updates the cached attributes of a file specified by a dentry object (usually invoked by the network filesystem).

The methods just listed are available to all possible inodes and filesystem types. However, only a subset of them applies to any specific inode and filesystem; the fields corresponding to unimplemented methods are set to NULL.

12.2.3 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.

The main information stored in a file object is the *file pointer*, that is, the current position in the file from which the next operation will take place. Since several processes may access the same file concurrently, the file pointer cannot be kept in the inode object.

Table 12-4. The Fields of the File Object

Type	Field	Description
`struct file *`	`f_next`	Pointer to next file object
`struct file **`	`f_pprev`	Pointer to previous file object
`struct dentry *`	`f_dentry`	Pointer to associated dentry object
`struct file_operations *`	`f_op`	Pointer to file operation table
`mode_t`	`f_mode`	Process access mode
`loff_t`	`f_pos`	Current file offset (file pointer)
`unsigned int`	`f_count`	File object's usage counter
`unsigned int`	`f_flags`	Flags specified when opening the file
`unsigned long`	`f_reada`	Read-ahead flag
`unsigned long`	`f_ramax`	Maximum number of pages to be read-ahead
`unsigned long`	`f_raend`	File pointer after last read-ahead
`unsigned long`	`f_ralen`	Number of read-ahead bytes
`unsigned long`	`f_rawin`	Number of read-ahead pages
`struct fown_struct`	`f_owner`	Data for asynchronous I/O via signals
`unsigned int`	`f_uid`	User's UID
`unsigned int`	`f_gid`	User's GID
`int`	`f_error`	Error code for network write operation
`unsigned long`	`f_version`	Version number, automatically incremented after each use
`void *`	`private_data`	Needed for tty driver

Each file object is always included in one of the following circular doubly linked lists:

- The list of "unused" file objects. This list acts both as a memory cache for the file objects and as a reserve for the superuser; it allows the superuser to open a file even if the dynamic memory in the system is exhausted. Since the objects are unused, their `f_count` fields are null. The address of the first element in the list is stored in the `free_filps` variable. The kernel makes sure that the list always contains at least `NR_RESERVED_FILES` objects, usually 10.
- The list of "in use" file objects. Each element in the list is used by at least one process, and hence its `f_count` field is not null. The address of the first element in the list is stored in the `inuse_filps` variable.

Regardless of which list a file object is in at the moment, its `f_next` field points to the next element in the list, while the `f_ pprev` field points to the `f_next` field of the previous element.

The size of the list of "unused" file objects is stored in the `nr_free_files` variable. The `get_empty_filp( )` function is invoked when the VFS must allocate a new file object. The function checks whether the "unused" list has more than `NR_RESERVED_FILES` items, in which case one can be used for the newly opened file. Otherwise, it falls back to normal memory allocation.

As we explained in Section 12.1.1, 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 default_file_ops field of the inode_operations structure 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:

llseek(file, offset, whence)

> 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 incremented.

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 incremented.

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.

mmap(file, vma)

> Performs a memory mapping of the file into a process address space (see Section 15.2 in Chapter 15).

open(inode, file)

> Opens a file by creating a new file object and linking it to the corresponding inode object (see Section 12.5.1 later in this chapter).

`flush(file)`

Called when a reference to an open file is closed, that is, the `f_count` field of the file object is decremented. 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, the `f_count` field of the file object becomes 0.

`fsync(file, dentry)`

Writes all cached data of the file to disk.

`fasync(file, on)`

Enables or disables asynchronous I/O notification by means of signals.

`check_media_change(dev)`

Checks whether there has been a change of media since the last operation on the device file (applicable to block devices that support removable media, such as floppies and CD-ROMs).

`revalidate(dev)`

Restores the consistency of a device (used by network filesystems after a media change has been recognized on a remote device).

`lock(file, cmd, file_lock)`

Applies a lock to the file (see Section 12.6 later in this chapter).

The methods just described are available to all possible file types. However, only a subset of them applies to a specific file type; the fields corresponding to unimplemented methods are set to `NULL`.

12.2.4 Special Handling for Directory File Objects

Directories must be handled with care because several processes can change their contents concurrently. Explicit locking, which is frequently performed on regular files (see Section 12.6 later in this chapter), is not well suited for directories because it prevents other processes from accessing the whole subtree of files rooted at the locked directory. Therefore, the `f_version` field of the file object is used together with the `i_version` field of the inode object to ensure that accesses to each directory file maintain consistency.

We'll explain the use of these fields by describing the most common operation in which they are needed, the `readdir( )` system call. Each invocation of this call is supposed to return a directory entry and update the directory's file pointer so that the next invocation of the same system call will return the next directory entry. But the directory could be modified by another process that concurrently accesses it. Without some kind of consistency check, the `readdir( )` system call could return the wrong directory entry. Long intervals—potentially hours—could elapse between a process's calls to `readdir( )`, and the process may choose to stop calling it at any time, so we don't want to lock the directory. What we want is a way to make `readdir( )` adapt to changes.

The problem is solved by introducing the `global_event` variable, which plays the role of version stamp. Whenever the inode object of a directory file is modified, the `global_event` is increased by 1, and the new version stamp is stored in the `i_version` field of the object. Whenever a file object is created or its file pointer is modified, `global_event` is increased by 1, and the new version stamp is stored in the `f_version` field of the object. When servicing the `readdir( )` system call, the VFS checks whether the version stamps contained in the `i_version` and `f_version` fields coincide. If not, the directory may have been modified by some other process after the previous execution of `readdir( )`.

When the `readdir( )` call detects this consistency problem, it recomputes the directory's file pointer by reading again the whole directory contents. The system call returns the directory entry immediately following the entry that was returned by the process's last `readdir( )`. `f_version` is then set to `i_version` to indicate that `readdir( )` is now synchronized with the actual state of the directory.

12.2.5 Dentry Objects

We mentioned in Section 12.1.1 that each directory is considered by the VFS as a normal file that contains a list of files and other directories. We shall discuss in Chapter 17 how directories are implemented on a specific filesystem. Once a directory entry has been 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. A dentry object is created by the kernel 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 called `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
`int`	`d_count`	Dentry object usage counter
`unsigned int`	`d_flags`	Dentry flags
`struct inode *`	`d_inode`	Inode associated with filename
`struct dentry *`	`d_parent`	Dentry object of parent directory
`struct dentry *`	`d_mounts`	For a mount point, the dentry of the root of the mounted filesystem
`struct dentry *`	`d_covers`	For the root of a filesystem, the dentry of the mount point
`struct list_head`	`d_hash`	Pointers for list in hash table entry
`struct list_head`	`d_lru`	Pointers for unused list
`struct list_head`	`d_child`	Pointers for the list of dentry objects included in parent directory
`struct list_head`	`d_subdirs`	For directories, list of dentry objects of subdirectories
`struct list_head`	`d_alias`	List of associated inodes (alias)
`struct qstr`	`d_name`	Filename
`unsigned long`	`d_time`	Used by `d_revalidate` method
`structdentry_operations*`	`d_op`	Dentry methods
`struct super_block *`	`d_sb`	Superblock object of the file
`unsigned long`	`d_reftime`	Time when dentry was discarded
`void *`	`d_fsdata`	Filesystem-dependent data
`unsigned char`	`d_iname[16]`	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 null, 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 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 no longer exists, because the corresponding disk inode has been deleted. 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 misleading since no negative value is involved.

12.2.6 The Dentry Cache

Since 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, then print or copy, then edit. In any case like these, the same file needs to be repeatedly accessed.

In order 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 hashing 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, since the dentry cache is still using them and therefore their `i_count` fields are not null. 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. 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 (since 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 Section 16.7.3 in Chapter 16).

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 depends on the amount of RAM installed in the system. 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 address of the dentry object of the directory and the filename.

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)`

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, hash)`

Creates a hash value; a filesystem-specific hash function for the dentry hash table. The `dentry` parameter identifies the directory containing the component. The `hash` 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.

12.2.7 Files Associated with a Process

We mentioned in Section 1.5 in Chapter 1 that each process has its own current working directory and its own root directory. This information is stored in an `fs_struct` kernel table, whose address is contained in the `fs` field of the process descriptor.

```
struct fs_struct {
    atomic_t count;
    int umask;
    struct dentry * root, * pwd;
};
```

The count field specifies the number of processes sharing the same fs_struct table (see Section 3.3.1 in Chapter 3). The umask field is used by the umask( ) system call to set initial file permissions on newly created files.

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 12-6. A process cannot have more than NR_OPEN (usually, 1024) file descriptors. It is possible to define a smaller, dynamic bound on the maximum number of allowed open files by changing the rlim[RLIMIT_NOFILE] structure in the process descriptor.

Table 12-6. The Fields of the files_struct Structure

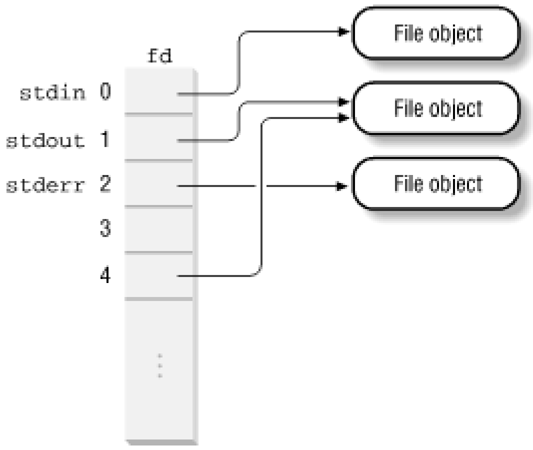
Type	Field	Description
int	count	Number of processes sharing this table
int	max_fds	Current maximum number of file objects
int	max_fdset	Current maximum number of file descriptors
int	next_fd	Maximum file descriptors ever allocated plus 1
struct file **	fd	Pointer to array of file object pointers
fd_set *	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[32]	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 12-4). 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 like 2>&1 to redirect the standard error to the standard output.

The open_fds field 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. Since the fd_set data structure includes 1024 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 12-4. The fd array



The kernel provides an fget( ) function to be invoked when it 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( ) increments by 1 the file object usage counter f_count.

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 decrements its usage counter f_count. Moreover, if this field becomes null, the function invokes the release method of the file operations (if defined), releases the associated dentry object, decrements the i_writeaccess field in the inode object (if the file was opened for writing), and finally moves the file object from the "in use" list to the "unused" one.

12.3 Filesystem Mounting

Now we'll focus on how the VFS keeps track of the filesystems it is supposed to support. Two basic operations must be performed before making use of a filesystem: registration and mounting.

Registration is done either when the system boots or when the module implementing the filesystem is being loaded. Once a filesystem has been registered, its specific functions are available to the kernel, so that kind of filesystem can be mounted on the system's directory tree.

Each filesystem has its own *root directory*. The filesystem whose root directory is the root of the system's directory tree is called *root filesystem*. Other filesystems can be mounted on the system's directory tree: the directories on which they are inserted are called *mount points*.

12.3.1 Filesystem Registration

Often, the user configures Linux to recognize all the filesystems needed when compiling the kernel for her 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 filesystems whose code is currently included in the kernel. It does this by performing *filesystem registrations*.

Each registered filesystem is represented as a `file_system_type` object whose fields are illustrated in Table 12-7. All filesystem-type objects are inserted into a simply linked list. The `file_systems` variable points to the first item.

Table 12-7. The Fields of the file_system_type Object

Type	Field	Description
`const char *`	`name`	Filesystem name
`int`	`fs_flag`	Mount flags
`struct super_block *(*)( )`	`read_super`	Method for reading superblock
`struct file_system_type *`	`next`	Pointer to next list element

During system initialization, the `filesystem_setup( )` function is invoked to register the filesystems specified at compile time. For each filesystem type, the `register_filesystem( )` function is invoked with a parameter pointing to the proper `file_system_type` object, which is thus inserted into the filesystem-type list.

The `register_filesystem( )` 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 and returns a pointer to the corresponding `file_system_type` object if it is present.

12.3.2 Mounting the Root Filesystem

Mounting the root filesystem is a crucial part of system initialization. While the system boots, it finds the major number of the disk containing the root filesystem in the `ROOT_DEV` variable. The root filesystem can be specified as a device file in the */dev* directory either when compiling the kernel or by passing a suitable 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 */sbin/rdev* external program on a compiled kernel image or by passing suitable options to the initial bootstrap loader (see Appendix A).

During system initialization, right after the `filesystem_setup( )` invocation, the `mount_root( )` function is executed. It performs the following operations (assuming that the filesystem to be mounted is a disk-based one):[4]

[4] Diskless workstations can mount the root directory over a network-based filesystem such as NFS, but we don't describe how this is done.

1. Initializes a dummy, local file object `filp`. The `f_mode` field is set according to the mount flags of the root, while all other fields are set to 0.
2. Creates a dummy inode object and initializes it by setting its `i_rdev` field to `ROOT_DEV`.
3. Invokes the `blkdev_open( )` function, passing the dummy inode and the file object. As we shall see later in Chapter 13, the function checks whether the disk exists and is properly working.
4. Releases the dummy inode object, which was needed just to verify the disk.
5. Scans the filesystem-type list. For each `file_system_type` object, invokes `read_super( )` to attempt to read the corresponding superblock. This function checks

that the device is not already mounted and attempts to fill a superblock object by using the method to which the read_super field of the file_system_type object points. Since each filesystem-specific method uses unique magic numbers, all read_super( ) invocations will fail except the one that attempts to fill the superblock by using the method of the filesystem really used on the root device. The read_super( ) method also creates an inode object and a dentry object for the root directory; the dentry object maps / to the inode object.

6. Sets the root and pwd fields of the fs_struct table of current (the *init* process) to the dentry object of the root directory.
7. Invokes add_vfsmnt( ) to insert a first element into the list of mounted filesystems (see next section).

12.3.3 Mounting a Generic Filesystem

Once the root filesystem has been initialized, additional filesystems may be mounted. Each of them must have its own mount point, which is just an already existing directory in the system's directory tree.

All mounted filesystems are included in a list, whose first element is referenced by the vfsmntlist variable. Each element is a structure of type vfsmount, whose fields are shown in Table 12-8.

Table 12-8. The Fields of the vfsmount Data Structure

Type	Field	Description
kdev_t	mnt_dev	Device number
char *	mnt_devname	Device name
char *	mnt_dirname	Mount point
unsigned int	mnt_flags	Device flags
struct super_block *	mnt_sb	Superblock pointer
struct quota_mount_options	mnt_dquot	Disk quota mount options
struct vfsmount *	mnt_next	Pointer to next list element

Three low-level functions are used to handle the list and are invoked by the service routines of the mount( ) and umount( ) system calls. The add_vfsmnt( ) and remove_vfsmnt( ) functions add and remove, respectively, an element in the list. The lookup_vfsmnt( ) function searches a specific mounted filesystem and returns the address of the corresponding vfsmount data structure.

The mount( ) system call is used to mount a 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 12-9)
- A pointer to a filesystem-dependent data structure (which may be NULL)

Table 12-9. Filesystem Mounting Options

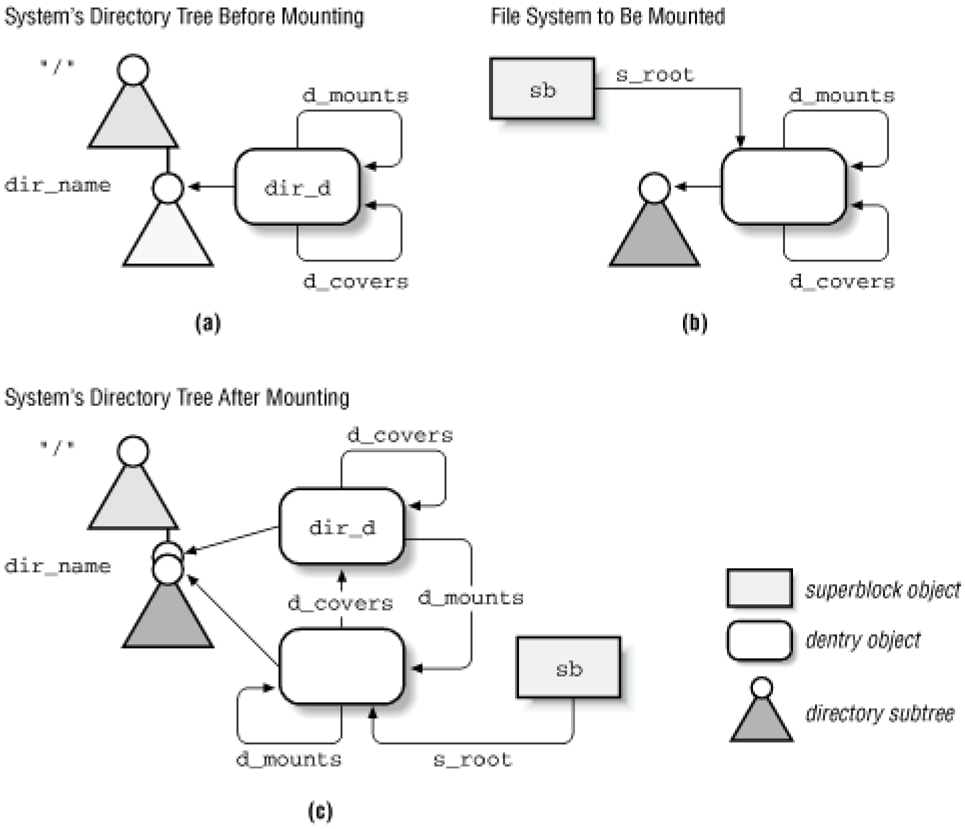
Macro	Value	Description
`MS_MANDLOCK`	`0x040`	Mandatory locking allowed.
`MS_NOATIME`	`0x400`	Do not update file access time.
`MS_NODEV`	`0x004`	Forbid access to device files.
`MS_NODIRATIME`	`0x800`	Do not update directory access time.
`MS_NOEXEC`	`0x008`	Disallow program execution.
`MS_NOSUID`	`0x002`	Ignore *setuid* and *setgid* flags.
`MS_RDONLY`	`0x001`	Files can only be read.
`MS_REMOUNT`	`0x020`	Remount the filesystem.
`MS_SYNCHRONOUS`	`0x010`	Write operations are immediate.
`S_APPEND`	`0x100`	Allow append-only file.
`S_IMMUTABLE`	`0x200`	Allow immutable file.
`S_QUOTA`	`0x080`	Initialize disk quota.

The `sys_mount( )` function performs the following operations:

1. Checks whether the process has the required capability to mount a filesystem.
2. If the `MS_REMOUNT` option has been specified, invokes `do_remount( )` to modify the mount flags and terminate.
3. Otherwise, gets a pointer to the proper `file_system_type` object by invoking `get_fs_type( )`.
4. If the filesystem to be mounted refers to a hardware device like */dev/hda1*, checks whether the device exists and is operational. This is done as follows:
    a. Invokes `namei( )` to get the dentry object of the corresponding device file (see the section Section 12.4 later in this chapter).
    b. Checks whether the inode associated with the device file refers to a valid block device (see Section 13.2.1 in Chapter 13).
    c. Initializes a dummy file object that refers to the device file, then opens the device file by using the `open` method of the file operations. If this operation succeeds, the device is operational.
5. If the filesystem to be mounted does not refer to a hardware device, gets a fictitious block device with major number by invoking `get_unnamed_dev( )`.
6. Invokes `do_mount( )`, passing the parameters `dev` (device number), `dev_name` (device filename), `dir_name` (mount point), `type` (filesystem type), `flags` (mount flags), and `data` (pointer to optional data area). This function mounts the required filesystem by performing the following operations:
    a. Invokes `namei( )` to locate the `dir_d` dentry object corresponding to `dir_name`; if it does not exist, creates it (see Figure 12-5 (a)).
    b. Acquires the `mount_sem` semaphore, which is used to serialize the mounting and unmounting operations.
    c. Checks to make sure that `dir_d->d_inode` is the inode of a directory and that the directory is not the root of a filesystem that is already mounted (`dir_d->d_covers` must be equal to `dir_d`).
    d. Invokes `read_super( )` to get the superblock object `sb` of the new filesystem. (If the object does not exist, it is created and filled with information read from the `dev` device.) The `s_root` field of the superblock object points to the dentry object of the root directory of the filesystem to be mounted (see Figure 12-5 (b)).

e. The previous operation could have suspended the current process; therefore, checks that no other process is using the superblock and that no process has already succeeded in mounting the same filesystem.
f. Invokes `add_vfsmnt( )` in order to insert a new element in the list of mounted filesystems.
g. Sets the `d_mounts` field of `dir_d` to the `s_root` field of the superblock, that is, to the root directory of the mounted filesystem.
h. Sets the `d_covers` field of the dentry object of the root directory of the mounted filesystem to `dir_d` (see Figure 12-5 (c)).
i. Releases the `mount_sem` semaphore.

Figure 12-5. Mounting a filesystem



Now, the `dir_d` dentry object of the mount point is linked through the `d_mounts` field to the root directory dentry object of the mounted filesystem; this latter object is linked to the `dir_d` dentry object through the `d_covers` field.

12.3.4 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 directory or a block device file) and a set of flags. It performs the following actions:

1. Checks whether the process has the required capability to unmount the filesystem.
2. Invokes `namei( )` on the filename to derive the `dentry` pointer to the associated dentry object.

3. If the filename refers to the mount point, derives the device identifier from dentry->d_inode->i_sb->s_dev. In other words, the function goes from the dentry object of the mount point to the relative inode, then to the corresponding superblock, and finally to the device identifier.
4. Otherwise, if the filename refers to the device file, derives the device identifier from dentry->d_inode->i_rdev.
5. Invokes dput(dentry) to release the dentry object.
6. Flushes the buffers of the device (see Section 14.1 in Chapter 14).
7. Acquires the mount_sem semaphore.
8. Invokes do_umount( ), which performs the following operations:
    a. Invokes get_super( ) to get the pointer sb of the superblock of the mounted filesystem.
    b. Invokes shrink_dcache_sb( ) to remove the dentry objects that refer to the dev device without disturbing other dentries. The dentry object of the root directory of the mounted filesystem will not be removed, since it is still used by the process doing the unmount.
    c. Invokes fsync_dev( ) to force all "dirty" buffers that refer to the dev device to be written to disk.
    d. If dev is the root device (dev == ROOT_DEV), it cannot be unmounted. If it has not been already remounted, remounts it with the MS_RDONLY flag set and returns.
    e. Checks whether the usage counter of the dentry object corresponding to the root directory of the filesystem to be unmounted is greater than 1. If so, some process is accessing a file in the filesystem, so returns an error code.
    f. Decrements the usage counter of sb->s_root->d_covers (the dentry object of the mount point directory).
    g. Sets sb->s_root->d_covers->d_mounts to sb->s_root->d_covers. This removes the link from the inode of the mount point to the inode of the root directory of the filesystem.
    h. Releases the dentry object to which sb->s_root (the root directory of the previously mounted filesystem) points and sets sb->s_root to NULL.
    i. If the superblock has been modified and the write_super superblock's method is defined, executes it.
    j. If defined, invokes the put_super( ) method of the superblock.
    k. Sets sb->s_dev to 0.
    l. Invokes remove_vfsmnt( ) to remove the proper element from the list of mounted filesystems.
9. Invokes fsync_dev( ) to force a write to disk for all remaining "dirty" buffers that refer to the dev device (presumably, the buffers containing the superblock information), then invokes the release( ) method of the device file operations.
10. Releases the mount_sem semaphore.

12.4 Pathname Lookup

We illustrate in this section how the VFS derives an inode from the corresponding file pathname. When a process must identify a file, it passes its file pathname to some VFS system call, such as open( ), mkdir( ), rename( ), stat( ), and so on.

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 inode of the initial directory, the code examines the entry matching the first name to derive the corresponding inode. Then the directory file having 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, since it keeps the most recently used dentry objects in memory. As we have seen 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 the disk.

However, things are not as simple as they look, since 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 that 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.

The `namei( )` and `lnamei( )` functions derive an inode from a pathname. The difference between them is that `namei( )` follows a symbolic link if it appears as the last component in a pathname without trailing slashes, while `lnamei( )` does not. Both functions delegate the heavy work by invoking the `lookup_dentry( )` function, which acts on three parameters: `name` points to a file pathname, `base` points to the dentry object of the directory from which to start searching, and `lookup_flags` is a bit array that includes the following flags:

`LOOKUP_FOLLOW`

> If the last component of the pathname is a symbolic link, interpret (follow) it. This flag is set when `lookup_dentry( )` is invoked by `namei( )` and cleared when it is invoked by `lnamei( )`.

`LOOKUP_DIRECTORY`

> The last component of the pathname must be a directory.

LOOKUP_SLASHOK

A trailing / in the pathname is allowed even if the last filename does not exist.

LOOKUP_CONTINUE

There are still filenames to be examined in the pathname. This flag is used internally by lookup_dentry( ).

The lookup_dentry( ) function is recursive, since it may end up invoking itself. Therefore, name could represent the still unresolved trailing portion of a complete pathname. In this case, base points to the dentry object of the last resolved pathname component. lookup_dentry( ) executes the following actions:

1. Examines both the first character of name and the value of base to identify the directory from which the search should start. Three cases can occur.
   - The first character of name is /: the pathname is an absolute pathname, thus base is set to current->fs->root.
   - The first character of name is different from "/" and base is NULL: the pathname is a relative pathname and base is set to current->fs->pwd.
   - The first character of name is different from "/" and base is not NULL: the pathname is a relative pathname and base is left unchanged. (This case should occur only when lookup_dentry( ) is recursively invoked.)
2. Gets the inode of the initial directory from base->d_inode.
3. Clears the LOOKUP_CONTINUE, LOOKUP_DIRECTORY, and LOOKUP_SLASHOK flags in lookup_flags.
4. Iteratively repeats the following procedure on each filename included in the path. If an error condition is encountered, exits from the cycle and returns a NULL dentry pointer, else returns the dentry pointer corresponding to the file pathname. At the start of each iteration, name points to the next filename to be examined and base points to the dentry object of the current directory.
   a. Checks whether the process is allowed to access the base directory (if defined, uses the permission method of inode).
   b. Computes a hash value from the first component name in name to be used in searching for the corresponding entry in the dentry cache. Moreover, if the base directory is in a filesystem that has its own d_hash( ) dentry hashing method, invokes base->d_op->d_hash( ) to compute the hash value based on the directory, the component name, and the previous hash value.
   c. Updates name so that it points to the first character of the next component name (if any), skipping any "/" separator.
   d. Sets the flag local variable to the value previously set in lookup_flags. Additionally, if the currently resolved component was followed by a trailing /, sets the LOOKUP_DIRECTORY flag (requiring a check on whether the component is a directory) and the LOOKUP_FOLLOW flag (interprets the component even if it is a symbolic link). Moreover, if there is a non-null component after the component currently resolved, sets the LOOKUP_CONTINUE flag.
   e. Invokes reserved_lookup( ) to perform the following actions:
      a. If the first component name is a single period (.), sets the dentry local variable to base.

b. If the first component name is a double period (`..`) and `base` is equal to `current->fs->root`, sets the `dentry` local variable to `base` (because the process is already in its root directory).

c. If the first component name is a double period (`..`) and `base` is not equal to `current->fs->root`, sets the `dentry` local variable to `base->d_covers->d_parent`. Usually, `d_covers` points to `base` itself and `dentry` is set to the directory that includes `base`; however, if the `base` directory is the root of a mounted filesystem, the `d_covers` field points to the inode of the mount point and `dentry` is set to the directory that includes the mount point.

If the first component name is neither (`.`) nor (`..`) invokes `cached_lookup( )`, passing as parameters `base` and the hash number previously derived. If the dentry hash table includes the required object, returns its address in `dentry`.

If the required dentry object is not in the dentry cache, invokes `real_lookup( )` to read the directory from disk and creates a new dentry object. This function, which acts on the `base` and `name` parameters, performs the following steps:

d. Gets the `i_sem` semaphore of the directory inode.

e. Reexecutes `cached_lookup( )`, since the required dentry object could have been inserted in the cache while the process was waiting for the directory semaphore.

f. We assume that the previous attempt failed. Invokes `d_alloc( )` to allocate a new dentry object.

g. Invokes the `lookup` method of the inode associated with the `base` directory to find the directory entry containing the required `name` and fills the new dentry object. This method is filesystem-dependent. We'll describe its Ext2 implementation in Chapter 17.

h. Releases the `i_sem` semaphore of the directory inode.

i. Returns the address of the new object in `dentry` or an error code if the entry was not found.

f. Invokes `follow_mount( )` to check whether the `d_mounts` field of `dentry` has the same value as `dentry`. If not, `dentry` is the mount point of a filesystem. In this case, the old `dentry` object is replaced by the one having the address in `dentry->d_mounts`.

g. Invokes `do_follow_link( )` to check whether `name` is a symbolic link. This function receives as its parameters `base`, `dentry`, and `flags` and executes the following steps:

a. If the `LOOKUP_FOLLOW` flag is not set, returns immediately. Since the flag is set by `lnamei( )`, this ensures that `lnamei( )` does not follow a symbolic link if it appears as the last component in a pathname without trailing slashes.

b. Checks whether `dentry->d_inode` contains the `follow_link` method. If not, the inode is not a symbolic link, so the function returns the `dentry` input parameter.

c. Invokes the `follow_link` inode method. This filesystem-dependent function reads the pathname associated with the symbolic link from the disk and recursively invokes `lookup_dentry( )` to resolve it. The function then returns a pointer to the dentry object referred by the symbolic link (we shall describe in Chapter 17 how symbolic links are handled by Ext2).

   Since `lookup_dentry( )` invokes `do_follow_link( )`, which may in turn invoke the `follow_link` inode method, which invokes, in turn, `lookup_dentry( )`, recursive cycles of function calls may be created. The `link_count` field of `current` is used to avoid endless recursive calls due to circular references inside the symbolic links. This field is incremented before each recursive execution of `follow_link( )` and decremented right after. If it reaches the value 5, `do_follow_link( )` terminates with an error code. Therefore, while there is no limit on the number of symbolic links in a pathname, the level of nesting of symbolic links can be at most five.

h. If everything went smoothly, `base` now points to the dentry object associated to the currently resolved component, so sets `inode` to `base->d_inode`.
i. If the `LOOKUP_DIRECTORY` flag of `flag` is not set, the currently resolved component is the last one in the file pathname, so returns the address in `base`. Note that `base` could point to a negative dentry object; that is, there might be no associated inode. This is fine for the lookup operation, since the last component must not be followed.
j. Otherwise, if `LOOKUP_DIRECTORY` is set, there is a slash after the currently resolved component. There are two cases to consider:
   - `inode` points to a valid inode object. In this case, checks that it is a directory by seeing whether the `lookup` method of the inode operations is defined; if not, returns an error code. Then either starts a new cycle iteration if the `LOOKUP_CONTINUE` flag in `flags` is set (meaning that the currently resolved component is not the last one) or returns the address in `base` (meaning that the component is the last one, even if it is followed by a trailing slash).
   - `inode` is `NULL` (meaning that `base` points to a negative dentry object). Returns `base` only if `LOOKUP_CONTINUE` is cleared and `LOOKUP_SLASHOK` is set; otherwise, returns an error code. Since a negative dentry object represents a file that was removed, it must not appear in the middle of a pathname lookup (which happens when `LOOKUP_CONTINUE` is set). Moreover, a negative dentry object must not appear as the last component in the pathname when a trailing slash is present, unless explicitly allowed by setting `LOOKUP_SLASHOK`.

12.5 Implementations of VFS System Calls

For the sake of brevity, we cannot discuss the implementation of all VFS system calls listed in Table 12-1. However, it could be useful to sketch out the implementation of a few system calls, just to show how VFS's data structures interact.

Let us reconsider the example proposed at the beginning of this chapter: a user issues a shell command that copies an MS-DOS file */floppy/TEST* in an Ext2 file */tmp/test*. The command shell invokes an external program like *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 {
    l = read(inf, buf, 4096);
    write(outf, buf, l);
} while (l);
close(outf);
close(inf);
```

Actually, the code of the real *cp* program is more complicated, since it must also check for possible error codes returned by each system call. In our example, we just focus our attention on the "normal" behavior of a copy operation.

12.5.1 The open( ) System Call

The `open( )` system call is serviced by the `sys_open( )` function, which receives as 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 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 will be 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 will be rewritten from scratch (`O_TRUNC` flag). Table 12-10 lists all flags of the `open( )` system call.

Table 12-10. The Flags of the open( ) System Call

Flag Name	Description
FASYNC	Asynchronous I/O notification via signals
O_APPEND	Write always at end of the file
O_CREAT	Create the file if it does not exist
O_DIRECTORY	Fail if file is not a directory
O_EXCL	With O_CREAT, fail if the file already exists
O_LARGEFILE	Large file (size greater than 2 GB)
O_NDELAY	Same as O_NONBLOCK
O_NOCTTY	Never consider the file as a controlling terminal
O_NOFOLLOW	Do not follow a trailing symbolic link in pathname
O_NONBLOCK	No system calls will block on the file
O_RDONLY	Open for reading
O_RDWR	Open for both reading and writing
O_SYNC	Synchronous write (block until physical write terminates)
O_TRUNC	Truncate the file
O_WRONLY	Open for writing

Let us describe the operation of the sys_open( ) function. It performs the following:

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. Invokes get_empty_filp( ) to get a new file object.
    b. Sets the f_flags and f_mode fields of the file object according to the values of the flags and modes parameters.
    c. Invokes open_namei( ), which executes the following operations:
        a. Invokes lookup_dentry( ) to interpret the file pathname and gets the dentry object associated with the requested file.
        b. Performs a series of checks to verify whether the process is permitted to open the file as specified by the values of the flags parameter. If so, returns the address of the dentry object; otherwise, returns an error code.
    d. If the access is for writing, checks the value of the i_writecount field of the inode object. A negative value means that the file has been memory-mapped, specifying that write accesses must be denied (see the section Section 15.2 in Chapter 15). In this case, returns an error code. Any other value specifies the number of processes that are actually writing into the file. In the latter case, increments the counter.
    e. Initializes the fields of the file object; in particular, sets the f_op field to the contents of the i_op->default_file_ops field of the inode object. This sets up all the right functions for future file operations.
    f. If the open method of the (default) file operations is defined, invokes it.
    g. Clears the O_CREAT, O_EXCL, O_NOCTTY, and O_TRUNC flags in f_flags.
    h. Returns the address of the file object.
4. Sets current->files->fd[fd] to the address of the file object.

5. Returns `fd` .

12.5.2 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( )` will transfer the data from the file into the buffer, while `write( )` will do the opposite. Both system calls return the number of bytes that were successfully transferred or -1 to signal an error condition.[5]

[5] 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 null 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 any data was 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( )` to derive from `fd` the address `file` of the corresponding file object and increments the usage counter `file->f_count`.
2. Checks whether the flags in `file->f_mode` allow the requested access (read or write operation).
3. Invokes `locks_verify_area( )` to check whether there are mandatory locks for the file portion to be accessed (see Section 12.6 later in this chapter).
4. If executing a write operation, acquires the `i_sem` semaphore included in the inode object. This semaphore forbids a process to write into the file while another process is flushing to disk buffers relative to the same file (see Section 14.1.5 in Chapter 14). It also forbids two processes to write into the same file at the same time. Notice that, unless the `O_APPEND` flag is set, POSIX does not require serialized file accesses: if a programmer wants exclusive access to a file, he must use a file lock (see next section). Thus, it is possible that a process is reading from a file while another process is writing to it.
5. Invokes either `file->f_op->read` or `file->f_op->write` to transfer the data. Both functions return the number of bytes that were actually transferred. As a side effect, the file pointer is properly updated.
6. Invokes `fput( )` to decrement the usage counter `file->f_count`.
7. Returns the number of bytes actually transferred.

12.5.3 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, since the copy operation has been completed.

The `close( )` system call receives as its parameter `fd` 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 19, 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 any mandatory lock on the file
    c. Invokes `fput( )` to release the file object
4. Returns the error code of the `flush` method (usually 0).

12.6 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, the 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). Since a process can choose to lock just 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 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, just the whole file.

Traditional System V variants provide the `lockf( )` system call, which is just 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.[6] A file is marked as a candidate for mandatory locking by setting its set-group bit (SGID) and clearing the group-execute permission bit. Since 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.

[6] 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.

Whether processes use advisory or mandatory locks, they can make use of both shared *read locks* and exclusive *write locks*. Any number of 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 (see Table 12-11).

Table 12-11. Whether a Lock Is Granted

	Requested Lock	Requested Lock
Current Locks	**Read**	**Write**
No lock	Yes	Yes
Read locks	Yes	No
Write lock	No	No

12.6.1 Linux File Locking

Linux supports all fashions of file locking: advisory and mandatory locks and the `fcntl( )`, `flock( )`, and the `lockf( )` system calls. However, the `lockf( )` system call is just a library wrapper routine, and therefore will not be discussed here.

Mandatory locks can be enabled and disabled on a per-filesystem basis using the `MS_MANDLOCK` flag of the `mount( )` system call. The default is to switch off mandatory locking: in this case, both `flock( )` and `fcntl( )` create advisory locks. When the flag is set, `flock( )` still produces advisory locks, while `fcntl( )` produces mandatory locks if the file has the set-group bit on and the group-execute bit off; it produces advisory locks otherwise.

Beside 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.

A lock produced by `fcntl( )` is of type `FL_POSIX`, while a lock produced by `flock( )` is of type `FL_LOCK`. These two types of locks may safely coexist, but neither one has any effect on the other. Therefore, a file locked through `fcntl( )` does not appear locked to `flock( )` and vice versa.

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 the child across a fork( ).

An FL_LOCK lock is always associated with a file object. When a lock is requested, the kernel replaces any other lock that refers to the same file object. 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_LOCK locks that refer to the file object are destroyed. However, there could be other FL_LOCK read locks set by other processes for the same file (inode), and they still remain active.

12.6.2 File-Locking Data Structures

The file_lock data structure represents file locks; its fields are shown in Table 12-12. All file_lock data structures are included in a doubly linked list. The address of the first element is stored in file_lock_table, while the fields fl_nextlink and fl_prevlink store the addresses of the adjacent elements in the list.

Table 12-12. The Fields of the file_lock Data Structure

Type	Field	Description
struct file_lock *	fl_next	Next element in inode list
struct file_lock *	fl_nextlink	Next element in global list
struct file_lock *	fl_prevlink	Previous element in global list
struct file_lock *	fl_nextblock	Next element in process list
struct file_lock *	fl_prevblock	Previous element in process list
struct files_struct *	fl_owner	Owner's files_struct
unsigned int	fl_pid	PID of the process owner
struct wait_queue *	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
off_t	fl_start	Starting offset of locked region
off_t	fl_end	Ending offset of locked region
void (*)(struct file_lock *)	fl_notify	Callback function when lock is unblocked
union	u	Filesystem-specific information

All lock_file structures that refer to the same file on disk are collected in a simply 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 tries to get an advisory or mandatory lock, it may be suspended until the previously allocated lock on the same file region is released. All processes sleeping on some lock are inserted into a wait queue, whose address is stored in the fl_wait field of the file_lock structure. Moreover, all processes sleeping on any file locks are inserted into a global circular list implemented by means of the fl_nextblock and fl_prevblock fields.

The following sections examine the differences between the two lock types.

12.6.3 FL_LOCK Locks

The flock( ) system call 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. If the LOCK_NB value is ORed to 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. Note that it is not possible to specify a region inside the file: the lock always applies to the whole file.

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 of the corresponding file object.
2. Invokes flock_make_lock( ) to initialize a file_lock structure by setting the fl_flags field to FL_LOCK; sets the fl_type field to F_RDLCK, F_WRLCK, or F_UNLCK, depending on the value of cmd, and sets the fl_file field to the address of the file object.
3. If the lock must be acquired, checks that the process has both read and write permission on the open file; if not, returns an error code.
4. Invokes flock_lock_file( ), passing as parameters the file object pointer filp, a pointer caller to the initialized file_lock structure, and a flag wait. This last parameter is set if the system call should block and cleared otherwise. This function performs, in turn, the following actions:
    a. Searches the list that filp->f_dentry->d_inode->i_flock points to. If an FL_LOCK lock for the same file object is found and an F_UNLCK operation is required, removes the file_lock element from the inode list and the global list, wakes up all processes sleeping in the lock's wait queue, frees the file_lock structure, and returns.
    b. Otherwise, searches the inode list again to verify that no existing FL_LOCK lock conflicts with the requested one. There must be no FL_LOCK write lock in the inode list, and moreover there must be no FL_LOCK lock at all if the processing is requesting a write lock. However, a process may want to change the type of a lock it already owns; this is done by issuing a second flock( ) system call. Therefore, the kernel always allows the process to change locks that refer to the same file object. If a conflicting lock is found and the LOCK_NB flag was specified, returns an error code, otherwise inserts the current process in the circular list of blocked processes and invokes interruptible_sleep_on( ) to suspend it.
    c. Otherwise, if no incompatibility exists, inserts the file_lock structure into the global lock list and the inode list, then returns (success).

12.6.4 FL_POSIX Locks

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 structure.

Locks of type FL_POSIX are able to protect an arbitrary file region, even a single byte. The region is specified by three fields of the flock structure. l_start is the initial offset of the region and is relative to the beginning of the file (if field l_whence is set to SEEK_SET), to the current file pointer (if l_whence is set to SEEK_CUR), or to the end of the file (if l_whence is set to SEEK_END). The l_len field specifies the length of the file region (or 0, which means that the region extends beyond the end of the file).

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 that 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.

When sys_fcntl( ) acquires a lock, it performs the following:

1. Reads the flock structure from user space.
2. Gets the file object corresponding to fd.
3. Checks whether the lock should be a mandatory one. In that case, returns an error code if the file has a shared memory mapping (see Section 15.2 in Chapter 15).
4. Invokes the posix_make_lock( ) function to initialize a new file_lock structure.
5. Returns an error code if the file does not allow the access mode specified by the type of the requested lock.
6. Invokes the lock method of the file operations, if defined.
7. Invokes the posix_lock_file( ) function, which executes the following actions:
    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 and fcntl( ) was invoked with the F_SETLK flag, returns an error code. Otherwise, the current process should be suspended. In this case, invokes posix_locks_deadlock( ) to check that no deadlock condition is being created among processes waiting for FL_POSIX locks, then

inserts the current process in the circular list of blocked processes and invokes `interruptible_sleep_on( )` to suspend it.

c. As soon as the inode's lock list includes no conflicting lock, 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 lock list and in the inode list.

8. Returns the value `0` (success).

12.7 Anticipating Linux 2.4

The Linux 2.4 VFS handles eight new filesystems, among them the `udf` for handling DVD devices. The maximum file size has been considerably increased (at least from the VFS point of view) by expanding the `i_size` field of the inode from 32 to 64 bits.

Additional access types can now be specified when opening a file: one refers to "raw" write requests that do not make use of the buffer cache.