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 (now owned by the SCO Group); the 4.4 BSD release from the University of California at Berkeley (4.4BSD); Digital UNIX from Digital Equipment Corporation (now Hewlett-Packard); AIX from IBM; HP-UX from Hewlett-Packard; Solaris from Sun Microsystems; and Mac OS X from Apple Computer, Inc. Beside Linux, a few other opensource Unix-like kernels exist, such as FreeBSD, NetBSD, and OpenBSD.

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

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

* LINUX® is a registered trademark of Linus Torvalds.

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

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

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

Linux Versus Other Unix-Like Kernels

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Hardware Dependency

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

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

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

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

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

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

i386
: IBM-compatible personal computers based on 80x86 microprocessors

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

m32r
: Computers based on the Renesas M32R family of microprocessors

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

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

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

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

s390
: IBM ESA/390 and zSeries mainframes

sh, sh64
: Embedded systems based on SuperH microprocessors developed by Hitachi and STMicroelectronics

sparc, sparc64
: Workstations based on Sun Microsystems SPARC and 64-bit Ultra SPARC microprocessors

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

v850
: NEC V850 microcontrollers that incorporate a 32-bit RISC core based on the Harvard architecture

x86_64
: Workstations based on the AMD's 64-bit microprocessors—such Athlon and Opteron—and Intel's ia32e/EM64T 64-bit microprocessors

Linux Versions

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

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

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

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

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

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

Basic Operating System Concepts

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

The operating system must fulfill two main objectives:

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

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

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

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

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

Multiuser Systems

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

Multiuser operating systems must include several features:

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

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

Users and Groups

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

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

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

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

Processes

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

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

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

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

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

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

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

Kernel Architecture

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

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

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

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

The main advantages of using modules include:

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

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

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

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

An Overview of the Unix Filesystem

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

Files

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

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

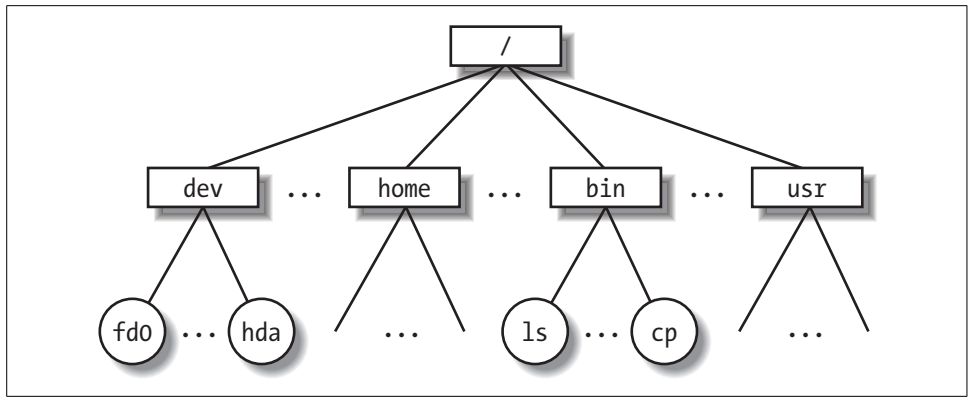


Figure 1-1. An example of a directory tree

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

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

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

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

Hard and Soft Links

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

The Unix command:

```
$ ln p1 p2
```

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

Hard links have two limitations:

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

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

The Unix command:

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

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

File Types

Unix files may have one of the following types:

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

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

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

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

File Descriptor and Inode

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

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

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

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

Access Rights and File Mode

The potential users of a file fall into three classes:

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

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

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

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

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

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

File-Handling System Calls

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

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

Opening a file

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

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

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

The three parameters have the following meanings:

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

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

`mode`
: Specifies the access rights of a newly created file.

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

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

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

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

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

Accessing an opened file

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

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

The `lseek( )` system call requires the following parameters:

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

which have the following meanings:

`fd`
: Indicates the file descriptor of the opened file

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

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

The `read( )` system call requires the following parameters:

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

which have the following meanings:

`fd`
: Indicates the file descriptor of the opened file

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

`count`
: Denotes the number of bytes to read

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

Closing a file

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

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

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

Renaming and deleting a file

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

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

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

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

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

An Overview of Unix Kernels

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

The Process/Kernel Model

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

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

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

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

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

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

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

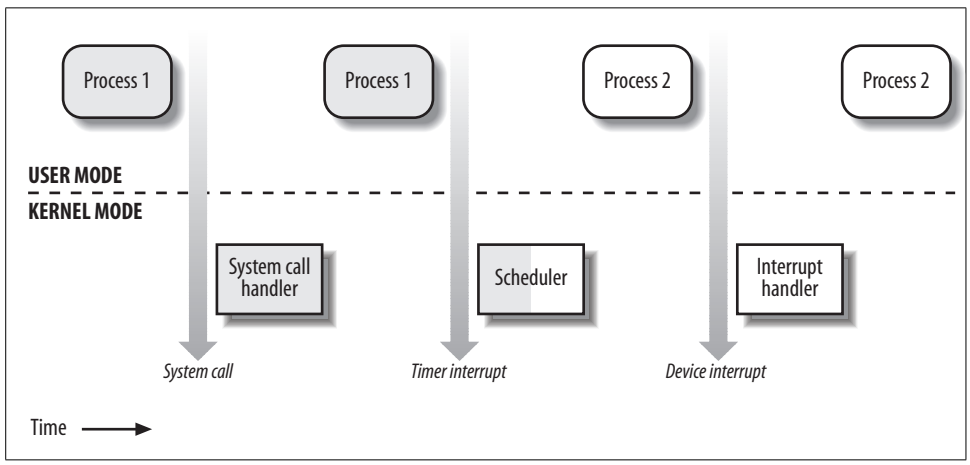


Figure 1-2. Transitions between User and Kernel Mode

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

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

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

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

Process Implementation

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

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

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

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

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

Reentrant Kernels

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

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

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

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

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

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

- A process executing in User Mode invokes a system call, and the corresponding kernel control path verifies that the request cannot be satisfied immediately; it then invokes the scheduler to select a new process to run. As a result, a process switch occurs. The first kernel control path is left unfinished, and the CPU resumes the execution of some other kernel control path. In this case, the two control paths are executed on behalf of two different processes.
- The CPU detects an exception—for example, access to a page not present in RAM—while running a kernel control path. The first control path is suspended, and the CPU starts the execution of a suitable procedure. In our example, this type of procedure can allocate a new page for the process and read its contents from disk. When the procedure terminates, the first control path can be resumed. In this case, the two control paths are executed on behalf of the same process.
- A hardware interrupt occurs while the CPU is running a kernel control path with the interrupts enabled. The first kernel control path is left unfinished, and the CPU starts processing another kernel control path to handle the interrupt. The first kernel control path resumes when the interrupt handler terminates. In this case, the two kernel control paths run in the execution context of the same process, and the total system CPU time is accounted to it. However, the interrupt handler doesn't necessarily operate on behalf of the process.
- An interrupt occurs while the CPU is running with kernel preemption enabled, and a higher priority process is runnable. In this case, the first kernel control path is left unfinished, and the CPU resumes executing another kernel control path on behalf of the higher priority process. This occurs only if the kernel has been compiled with kernel preemption support.

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

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

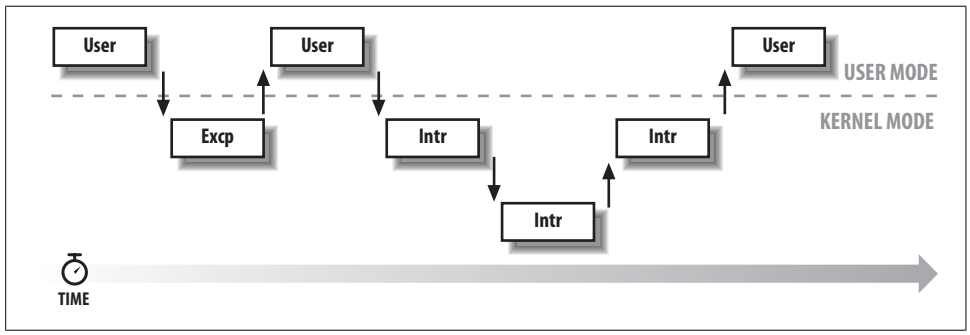


Figure 1-3. Interleaving of kernel control paths

Process Address Space

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

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

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

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

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

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

Synchronization and Critical Regions

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

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

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

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

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

Kernel preemption disabling

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

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

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

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

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

Interrupt disabling

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

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

Semaphores

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

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

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

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

Spin locks

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

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

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

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

Avoiding deadlocks

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

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

Signals and Interprocess Communication

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Process Management

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

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

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

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

Zombie processes

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

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

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

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

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

Process groups and login sessions

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

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

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

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

Memory Management

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

Virtual memory

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

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

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

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

Random access memory usage

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

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

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

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

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

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

Kernel Memory Allocator

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

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

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

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

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

Process virtual address space handling

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

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

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

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

Caching

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

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

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

Device Drivers

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

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

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

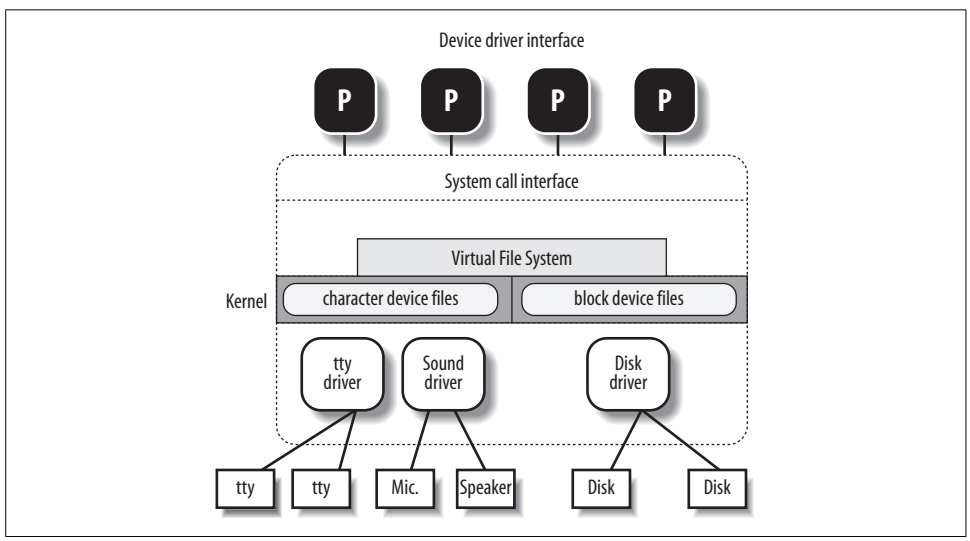


Figure 1-4. Device driver interface

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

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

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 so that programming errors cannot cause improper accesses to memory outside the program.

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

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

Memory Addresses

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

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

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

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

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

Figure 2-1. Logical address translation

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

Segmentation in Hardware

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

Segment Selectors and Segmentation Registers

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

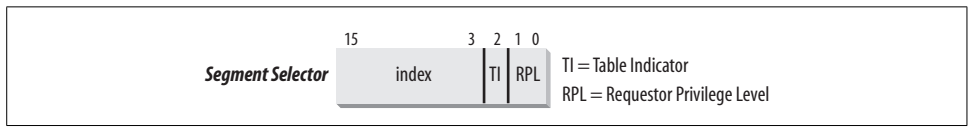


Figure 2-2. Segment Selector format

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

Three of the six segmentation registers have specific purposes:

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

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

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

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

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

Segment Descriptors

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

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

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

Table 2-1. Segment Descriptor fields

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

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

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

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

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

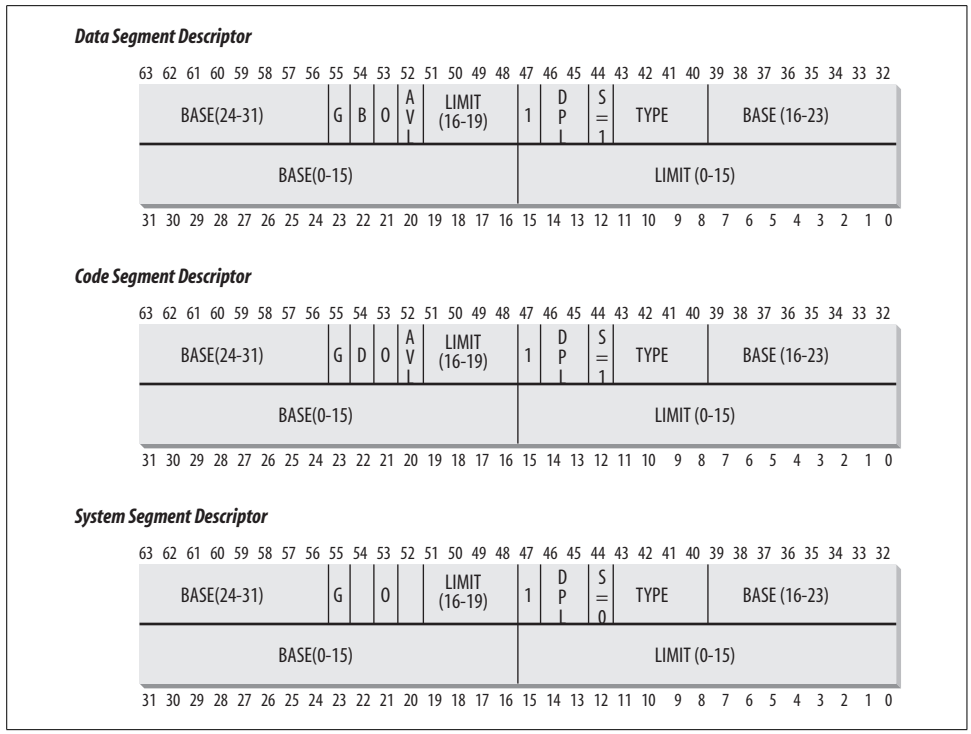


Figure 2-3. Segment Descriptor format

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

Fast Access to Segment Descriptors

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

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

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

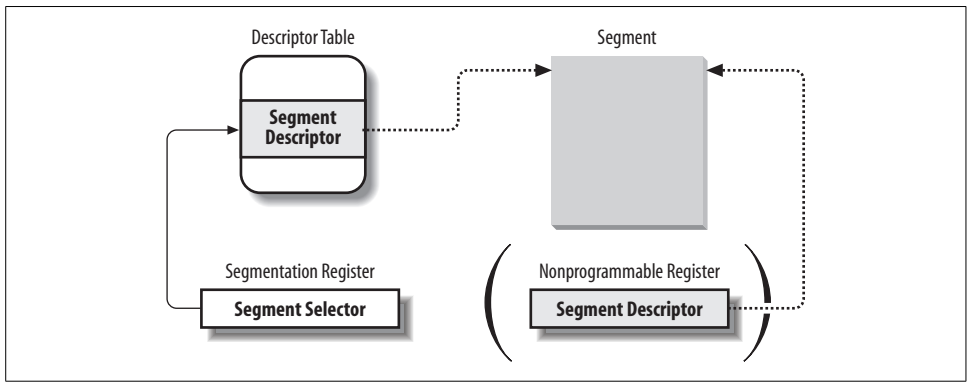


Figure 2-4. Segment Selector and Segment Descriptor

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

Table 2-2. Segment Selector fields

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

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

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

Segmentation Unit

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

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

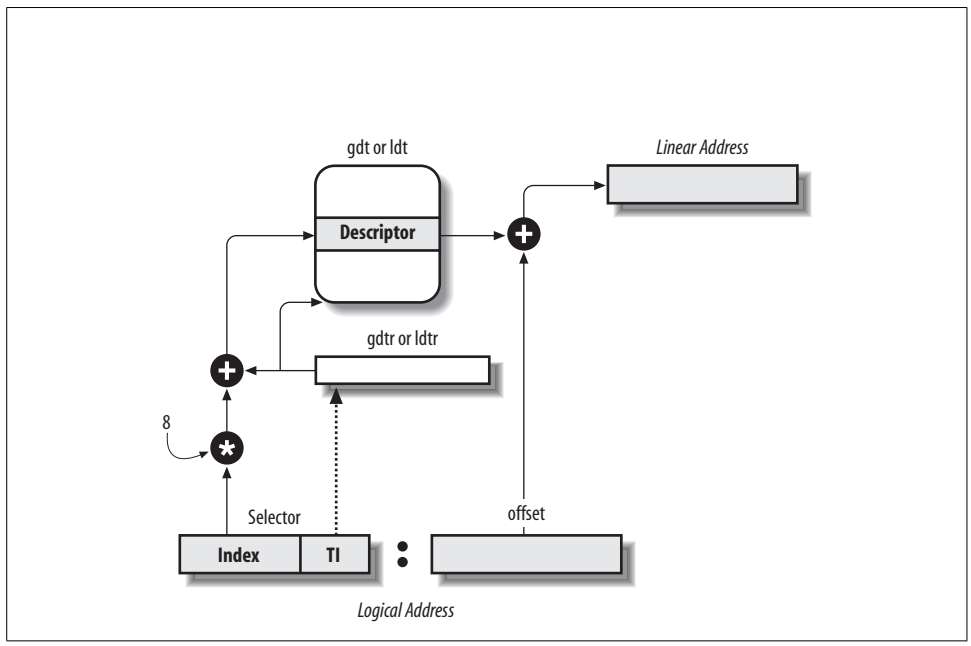


Figure 2-5. Translating a logical address

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

Segmentation in Linux

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

to separate the physical address spaces of processes: segmentation can assign a 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 a wide range of architectures; RISC architectures in particular have limited support for segmentation.

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

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

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

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

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

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

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

As stated earlier, the Current Privilege Level of the CPU indicates 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 CPL is changed, some segmentation registers must be correspondingly updated. For instance, when the `CPL` is equal to 3 (User Mode), the `ds` register must contain the Segment Selector of the user data segment,

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

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

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

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

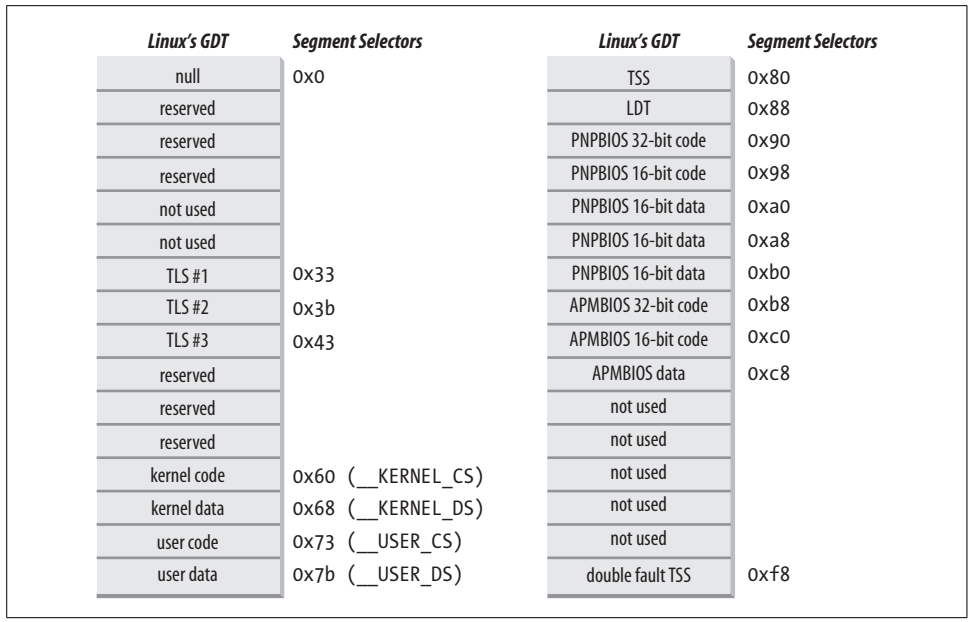
The Linux GDT

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

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

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

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



Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	Segment Selectors
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

Figure 2-6. The Global Descriptor Table

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

- A segment including the default Local Descriptor Table (LDT), usually shared by all processes (see the next section).
- Three *Thread-Local Storage (TLS)* segments: this is a mechanism that allows multithreaded applications to make use of up to three segments containing data local to each thread. The `set_thread_area()` and `get_thread_area()` system calls, respectively, create and release a TLS segment for the executing process.
- Three segments related to Advanced Power Management (APM): the BIOS code makes use of segments, so when the Linux APM driver invokes BIOS functions to get or set the status of APM devices, it may use custom code and data segments.
- Five segments related to Plug and Play (PnP) BIOS services. As in the previous case, the BIOS code makes use of segments, so when the Linux PnP driver invokes BIOS functions to detect the resources used by PnP devices, it may use custom code and data segments.

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

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

The Linux LDTs

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

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

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

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

Paging in Hardware

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

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

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

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

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

Regular Paging

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

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

Directory
: The most significant 10 bits

Table
: The intermediate 10 bits

Offset
: The least significant 12 bits

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

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

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

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

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

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

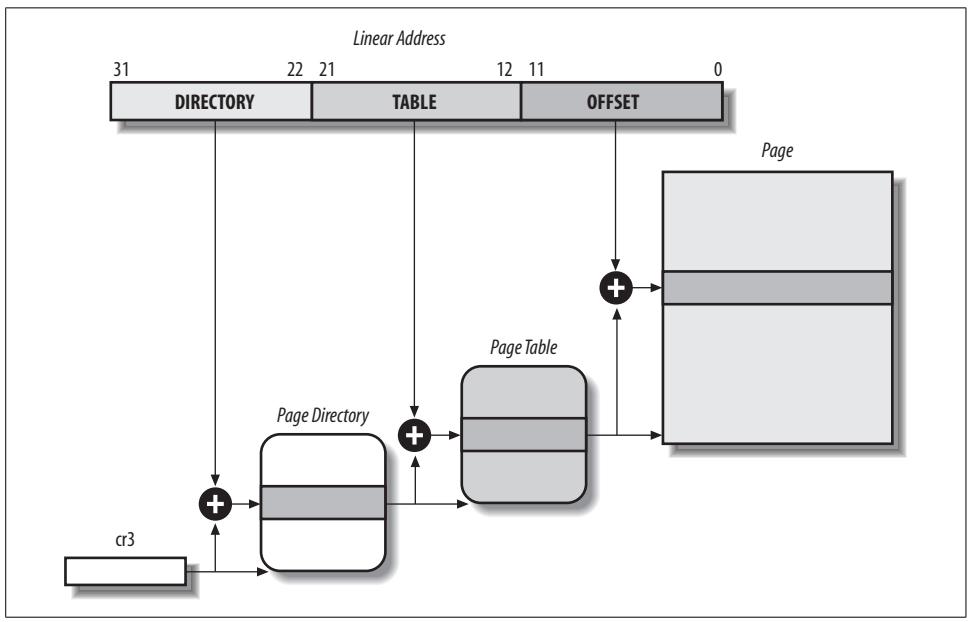


Figure 2-7. Paging by 80×86 processors

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

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

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

Field containing the 20 most significant bits of a page frame physical address
: Because each page frame has a 4-KB capacity, its physical address must be a multiple of 4096, so the 12 least significant bits of the physical address are always equal to 0. If the field refers to a Page Directory, the page frame contains a Page Table; if it refers to a Page Table, the page frame contains a page of data.

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

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

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

`User/Supervisor` *flag*
: Contains the privilege level required to access the page or Page Table (see the later section "Hardware Protection Scheme").

`PCD` *and* `PWT` *flags*
: Controls the way the page or Page Table is handled by the hardware cache (see the section "Hardware Cache" later in this chapter).

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

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

Extended Paging

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

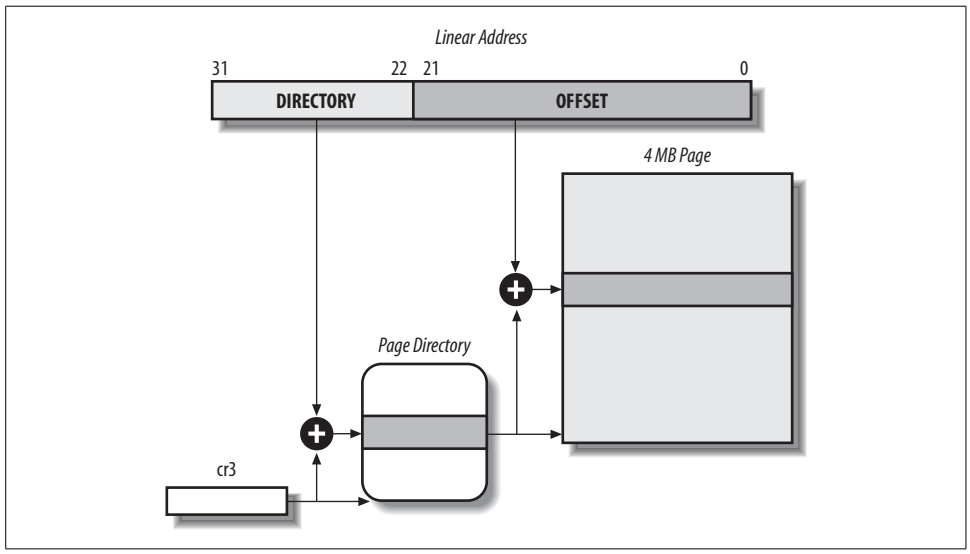


Figure 2-8. Extended paging

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

Directory
: The most significant 10 bits

Offset
: The remaining 22 bits

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

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

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

Hardware Protection Scheme

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

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

An Example of Regular Paging

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

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

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

† As we shall see in the following chapters, the 3 GB linear address space is an upper limit, but a User Mode process is allowed to reference only a subset of it.

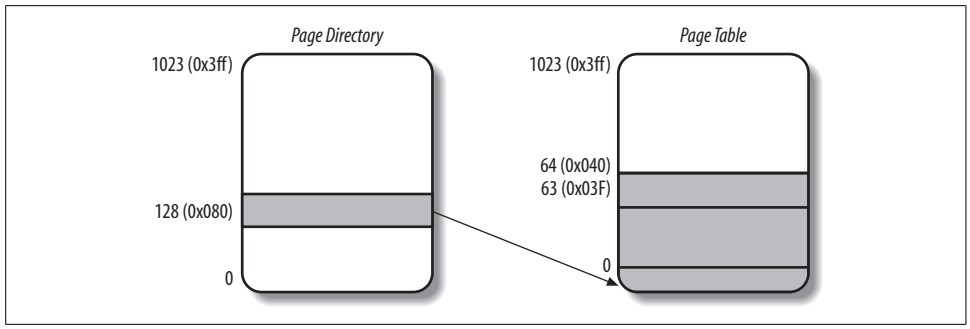


Figure 2-9. An example of paging

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

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

1. The Directory field `0x80` is used to select entry `0x80` of the Page Directory, which points to the Page Table associated with the process's pages.
2. The Table field `0x21` is used to select entry `0x21` of the Page Table, which points to the page frame containing the desired page.
3. Finally, the Offset field `0x406` is used to select the byte at offset `0x406` in the desired page frame.

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

The Physical Address Extension (PAE) Paging Mechanism

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

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

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

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

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

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

- The 64 GB of RAM are split into 2^{24} distinct page frames, and the physical address field of Page Table entries has been expanded from 20 to 24 bits. Because a PAE Page Table entry must include the 12 flag bits (described in the earlier section "Regular Paging") and the 24 physical address bits, for a grand total of 36, the Page Table entry size has been doubled from 32 bits to 64 bits. As a result, a 4-KB PAE Page Table includes 512 entries instead of 1,024.
- A new level of Page Table called the Page Directory Pointer Table (PDPT) consisting of four 64-bit entries has been introduced.
- The `cr3` control register contains a 27-bit Page Directory Pointer Table base address field. Because PDPTs are stored in the first 4 GB of RAM and aligned to a multiple of 32 bytes (2^5), 27 bits are sufficient to represent the base address of such tables.
- When mapping linear addresses to 4 KB pages (`PS` flag cleared in Page Directory entry), the 32 bits of a linear address are interpreted in the following way:

  `cr3`
  : Points to a PDPT

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

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

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

  bits 11–0
  : Offset of 4-KB page

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

  cr3
  : Points to a PDPT

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

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

  bits 20–0
  : Offset of 2-MB page

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

Paging for 64-bit Architectures

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

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

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

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

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

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

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

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

Hardware Cache

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

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

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

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

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

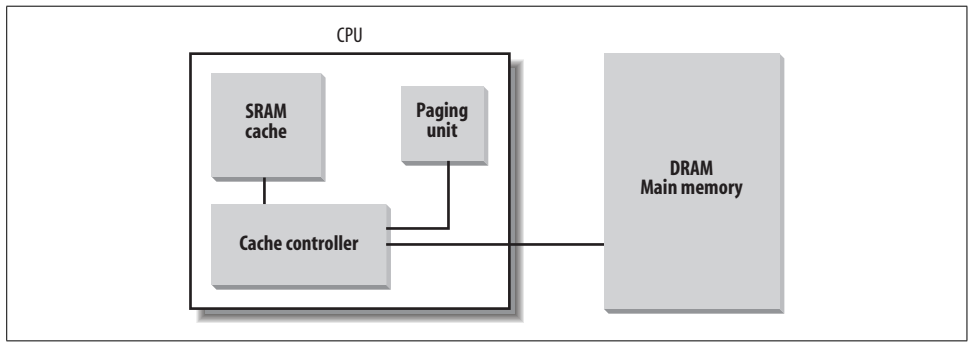


Figure 2-10. Processor hardware cache

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

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

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

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

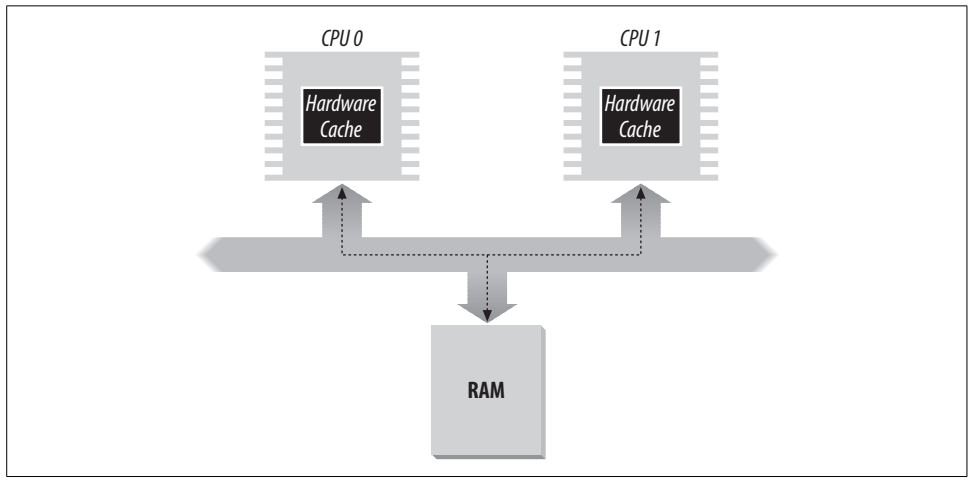


Figure 2-11. The caches in a dual processor

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

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

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

Translation Lookaside Buffers (TLB)

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

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

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

Paging in Linux

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

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

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

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

* This change has been made to fully support the linear address bit splitting used by the x86_64 platform (see Table 2-4).

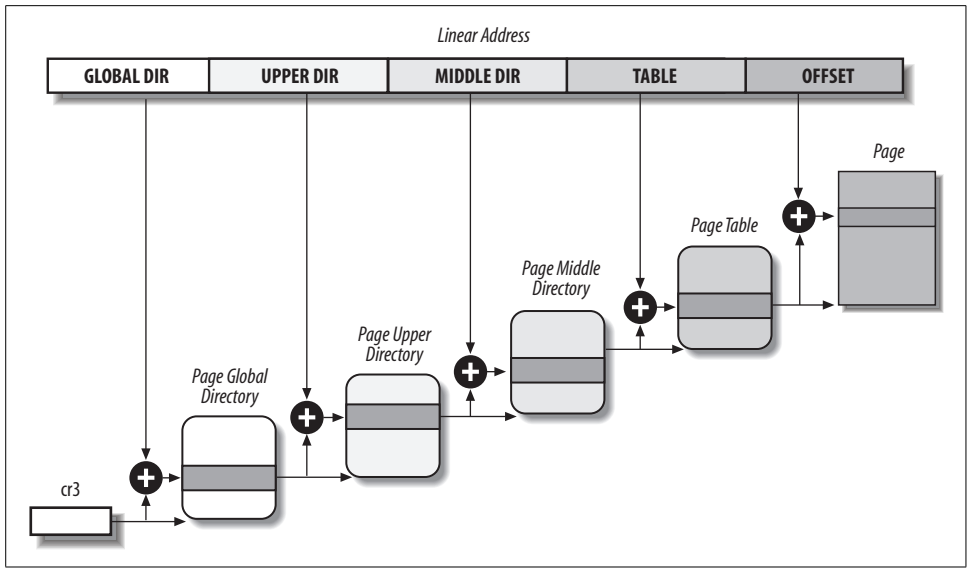


Figure 2-12. The Linux paging model

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

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

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

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

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

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

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

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

The Linear Address Fields

The following macros simplify Page Table handling:

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

PMD_SHIFT
: The total length in bits of the Offset and Table fields of a linear address; in other words, the logarithm of the size of the area a Page Middle Directory entry can map. The PMD_SIZE macro computes the size of the area mapped by a single entry of the Page Middle Directory—that is, of a Page Table. The PMD_MASK macro is used to mask all the bits of the Offset and Table fields.

  When PAE is disabled, PMD_SHIFT yields the value 22 (12 from Offset plus 10 from Table), PMD_SIZE yields 2^{22} or 4 MB, and PMD_MASK yields 0xffc00000. Conversely, when PAE is enabled, PMD_SHIFT yields the value 21 (12 from Offset plus 9 from Table), PMD_SIZE yields 2^{21} or 2 MB, and PMD_MASK yields 0xffe00000.

  Large pages do not make use of the last level of page tables, thus LARGE_PAGE_SIZE, which yields the size of a large page, is equal to PMD_SIZE (2PMD_SHIFT) while LARGE_PAGE_MASK, which is used to mask all the bits of the Offset and Table fields in a large page address, is equal to PMD_MASK.

PUD_SHIFT
: Determines the logarithm of the size of the area a Page Upper Directory entry can map. The PUD_SIZE macro computes the size of the area mapped by a single entry of the Page Global Directory. The PUD_MASK macro is used to mask all the bits of the Offset, Table, Middle Air, and Upper Air fields.

  On the 80×86 processors, PUD_SHIFT is always equal to PMD_SHIFT and PUD_SIZE is equal to 4 MB or 2 MB.

PGDIR_SHIFT
: Determines the logarithm of the size of the area that a Page Global Directory entry can map. The PGDIR_SIZE macro computes the size of the area mapped by a single entry of the Page Global Directory. The PGDIR_MASK macro is used to mask all the bits of the Offset, Table, Middle Air, and Upper Air fields.

  When PAE is disabled, PGDIR_SHIFT yields the value 22 (the same value yielded by PMD_SHIFT and by PUD_SHIFT), PGDIR_SIZE yields 2^{22} or 4 MB, and PGDIR_MASK yields 0xffc00000. Conversely, when PAE is enabled, PGDIR_SHIFT yields the value 30 (12 from Offset plus 9 from Table plus 9 from Middle Air), PGDIR_SIZE yields 2^{30} or 1 GB, and PGDIR_MASK yields 0xc0000000.

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

Page Table Handling

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

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

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

- pte_none, pmd_none, pud_none, and pgd_none yield the value 1 if the corresponding entry has the value 0; otherwise, they yield the value 0.
- pte_clear, pmd_clear, pud_clear, and pgd_clear clear an entry of the corresponding page table, thus forbidding a process to use the linear addresses mapped by the page table entry. The ptep_get_and_clear() function clears a Page Table entry and returns the previous value.

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

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

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

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

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

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

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

Table 2-5. Page flag reading functions

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

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

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

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

Table 2-6. Page flag setting functions

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

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

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

Table 2-7. Macros acting on Page Table entries

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

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

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

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

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

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

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

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

Table 2-8. Page allocation functions

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

Table 2-8. Page allocation functions (continued)

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

Physical Memory Layout

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

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

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

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

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

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

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

- Page frame 0 is used by BIOS to store the system hardware configuration detected during the *Power-On Self-Test* (*POST*); the BIOS of many laptops, moreover, writes data on this page frame even after the system is initialized.
- Physical addresses ranging from `0x000a0000` to `0x000fffff` are usually reserved to BIOS routines and to map the internal memory of ISA graphics cards. This area is the well-known hole from 640 KB to 1 MB in all IBM-compatible PCs: the physical addresses exist but they are reserved, and the corresponding page frames cannot be used by the operating system.
- Additional page frames within the first megabyte may be reserved by specific computer models. For example, the IBM ThinkPad maps the `0xa0` page frame into the `0x9f` one.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

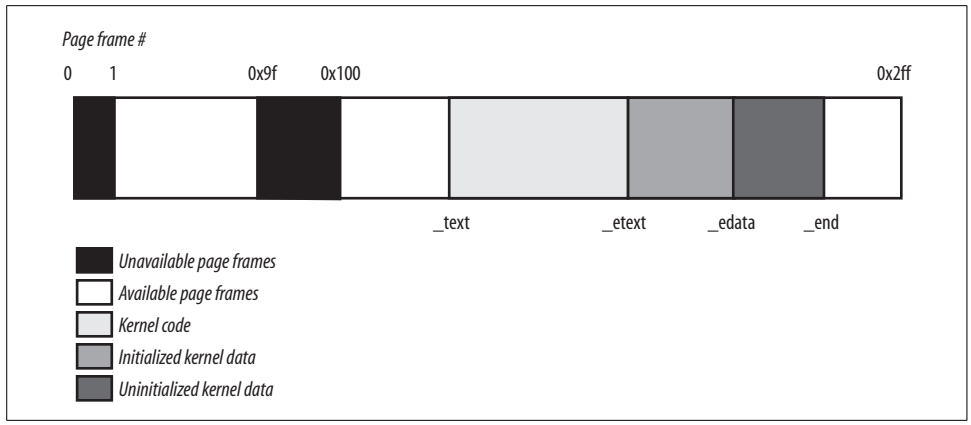


Figure 2-13. The first 768 page frames (3 MB) in Linux 2.6

Process Page Tables

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

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

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

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

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

Kernel Page Tables

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

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

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

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

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

Provisional kernel Page Tables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fix-Mapped Linear Addresses

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

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

Basically, a *fix-mapped linear address* is a constant linear address like 0xffffc000 whose corresponding physical address does not have to be the linear address minus 0xc0000000, but rather a physical address set in an arbitrary way. Thus, each fix-mapped linear address maps one page frame of the physical memory. As we'll see in later chapters, the kernel uses fix-mapped linear addresses instead of pointer variables that never change their value.

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

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

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

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

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

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

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

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

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

Handling the Hardware Cache and the TLB

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

Handling the hardware cache

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

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

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

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

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

Handling the TLB

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

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

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

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

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

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

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

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

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

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

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

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

- When performing a process switch between two regular processes that use the same set of page tables (see the section "The schedule() Function" in Chapter 7).
- When performing a process switch between a regular process and a kernel thread. In fact, we'll see in the section "Memory Descriptor of Kernel Threads" in Chapter 9, that kernel threads do not have their own set of page tables; rather, they use the set of page tables owned by the regular process that was scheduled last for execution on the CPU.

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

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

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

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

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

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

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

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

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

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

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* or *threads* in the Linux source code.

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

Processes, Lightweight Processes, and Threads

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

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

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

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

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

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

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

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

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

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

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

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

Process Descriptor

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

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

Process State

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

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

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

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

* The kernel also defines the task_t data type to be equivalent to struct task_struct.

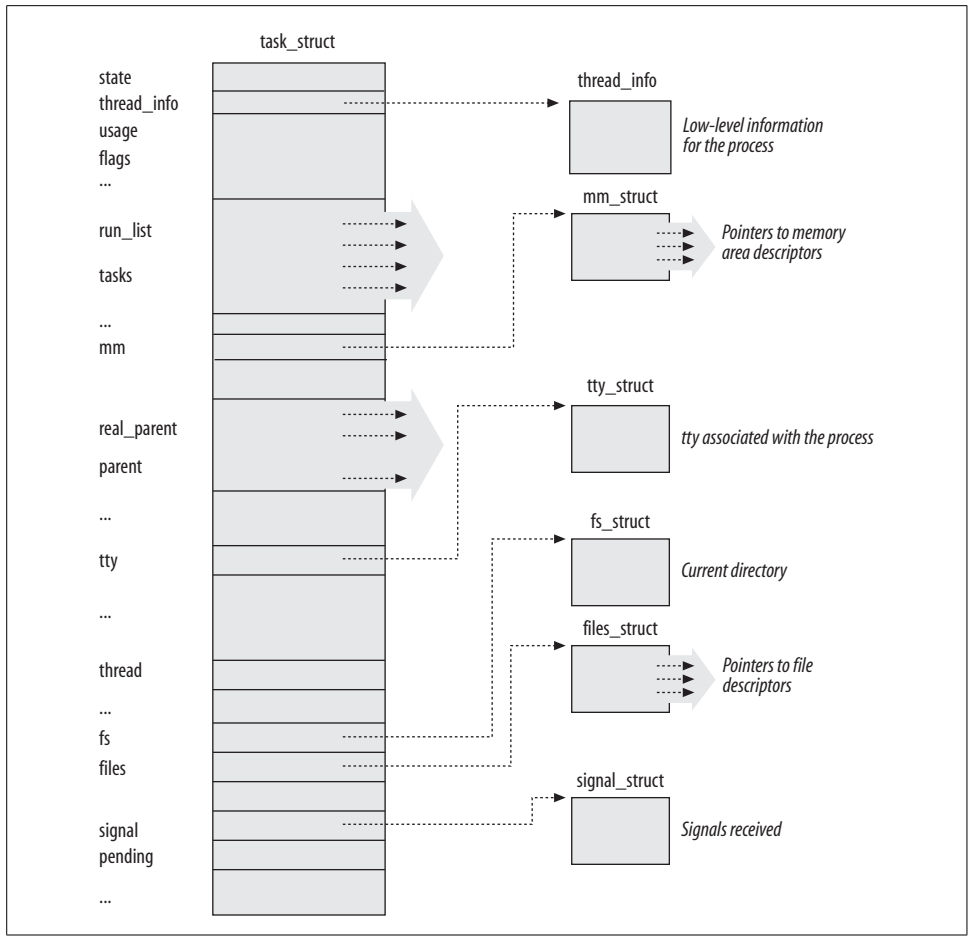


Figure 3-1. The Linux process descriptor

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

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

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

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

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

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

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

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

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

Identifying a Process

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

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

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

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

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

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

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

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

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

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

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

Process descriptors handling

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

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

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

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

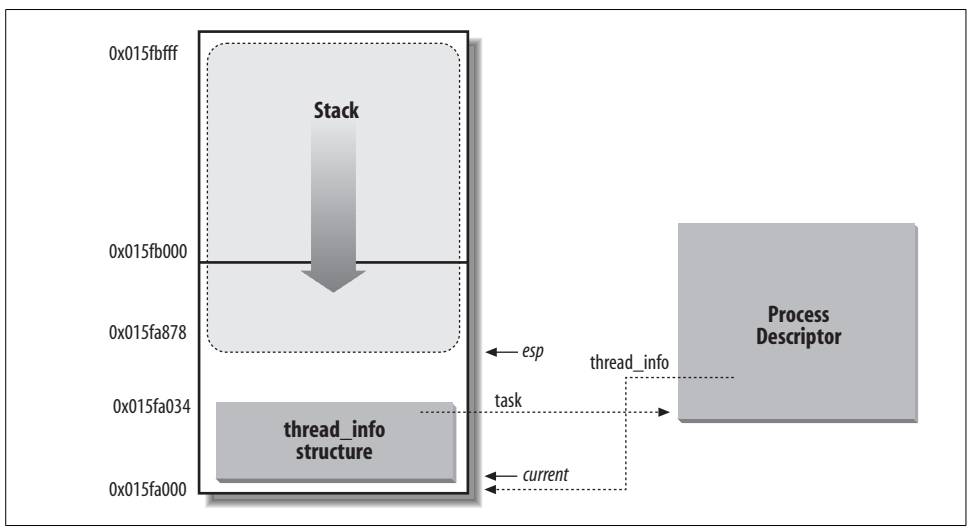


Figure 3-2. Storing the thread_info structure and the process kernel stack in two page frames

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

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

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

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

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

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

Identifying the current process

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

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

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

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

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

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

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

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

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

Doubly linked lists

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

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

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

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

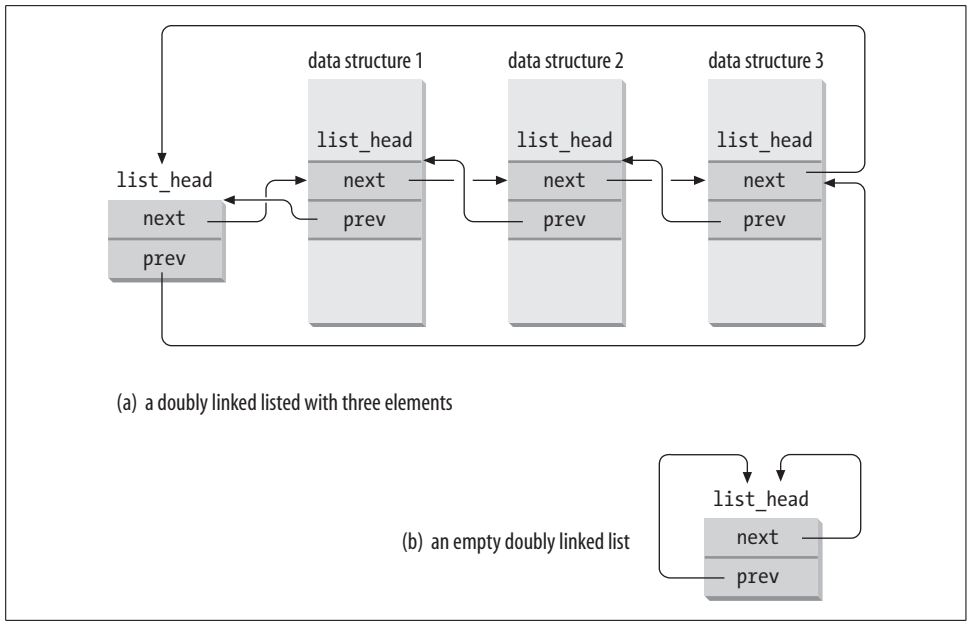


Figure 3-3. Doubly linked lists built with list_head data structures

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

Table 3-1. List handling functions and macros

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

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

The process list

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

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

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

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

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

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

The lists of TASK_RUNNING processes

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

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

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

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

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

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

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

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

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

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

Relationships Among Processes

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

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

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

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

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

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

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

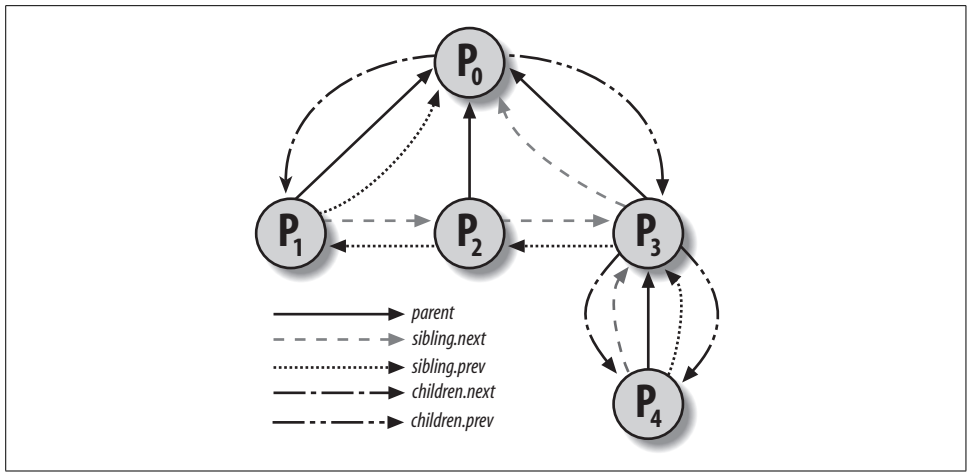


Figure 3-4. Parenthood relationships among five processes

The pidhash table and chained lists

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

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

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

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

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

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

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

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

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

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

The Magic Constant

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

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

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

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

The data structures used in the PID hash tables are quite sophisticated, because they must keep track of the relationships between the processes. As an example, suppose

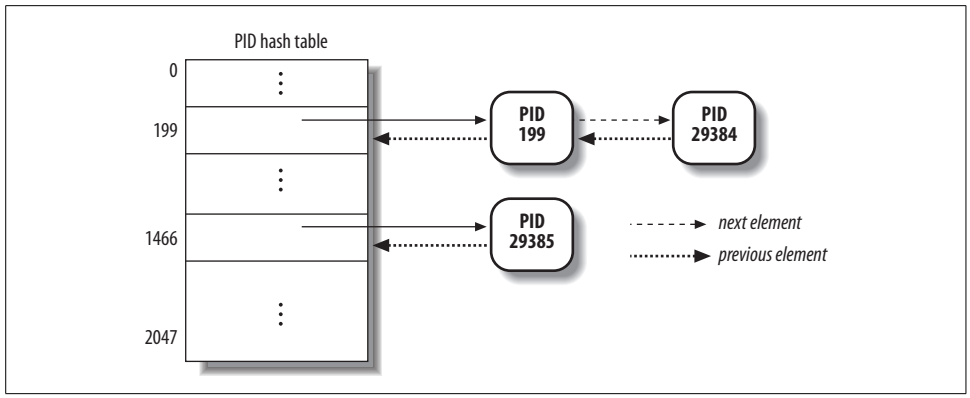


Figure 3-5. A simple PID hash table and chained lists

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

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

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

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

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

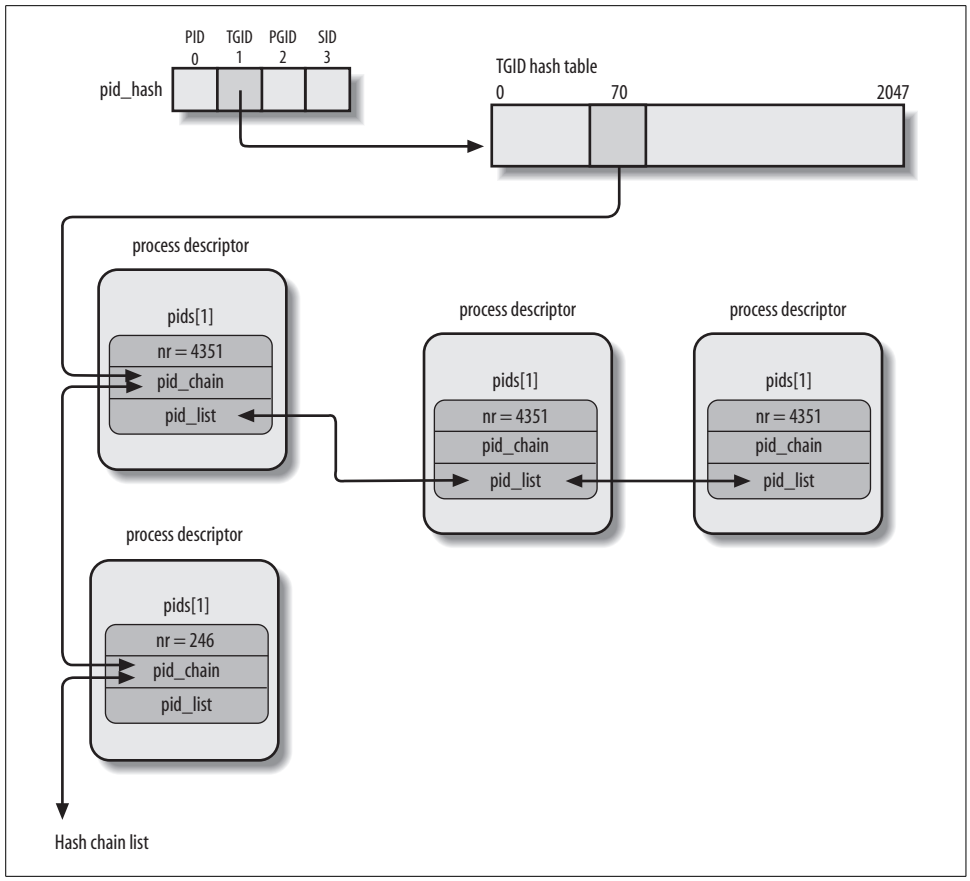


Figure 3-6. The PID hash tables

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

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

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

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

`find_task_by_pid(nr)`
: Same as `find_task_by_pid_type(PIDTYPE_PID, nr)`.

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

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

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

How Processes Are Organized

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

- Processes in a `TASK_STOPPED`, `EXIT_ZOMBIE`, or `EXIT_DEAD` state are not linked in specific lists. There is no need to group processes in any of these three states, because stopped, zombie, and dead processes are accessed only via PID or via linked lists of the child processes for a particular parent.
- Processes in a `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state are subdivided into many classes, each of which corresponds to a specific event. In this case, the process state does not provide enough information to retrieve the process quickly, so it is necessary to introduce additional lists of processes. These are called *wait queues* and are discussed next.

Wait queues

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

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

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

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

Elements of a wait queue list are of type `wait_queue_t`:

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

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

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

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

Handling wait queues

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

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

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

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

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

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

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

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

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

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

- The interruptible_sleep_on() function is identical to sleep_on(), except that it sets the state of the current process to TASK_INTERRUPTIBLE instead of setting it to TASK_UNINTERRUPTIBLE, so that the process also can be woken up by receiving a signal.
- The sleep_on_timeout() and interruptible_sleep_on_timeout() functions are similar to the previous ones, but they also allow the caller to define a time interval after which the process will be woken up by the kernel. To do this, they invoke the schedule_timeout() function instead of schedule() (see the section "An Application of Dynamic Timers: the nanosleep() System Call" in Chapter 6).
- The prepare_to_wait(), prepare_to_wait_exclusive(), and finish_wait() functions, introduced in Linux 2.6, offer yet another way to put the current process to sleep in a wait queue. Typically, they are used as follows:

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

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

  As soon as the process is awakened, it executes the finish_wait() function, which sets again the process state to TASK_RUNNING (just in case the awaking condition becomes true before invoking schedule()), and removes the wait queue element from the wait queue list (unless this has already been done by the wake-up function).
- The wait_event and wait_event_interruptible macros put the calling process to sleep on a wait queue until a given condition is verified. For instance, the wait_event(wq,condition) macro essentially yields the following fragment:

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

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

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

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

- All macros take into consideration sleeping processes in the TASK_INTERRUPTIBLE state; if the macro name does not include the string "interruptible," sleeping processes in the TASK_UNINTERRUPTIBLE state also are considered.
- All macros wake all nonexclusive processes having the required state (see the previous bullet item).
- The macros whose name include the string "nr" wake a given number of exclusive processes having the required state; this number is a parameter of the macro. The macros whose names include the string "all" wake all exclusive processes having the required state. Finally, the macros whose names don't include "nr" or "all" wake exactly one exclusive process that has the required state.
- The macros whose names don't include the string "sync" check whether the priority of any of the woken processes is higher than that of the processes currently running in the systems and invoke schedule() if necessary. These checks are not made by the macro whose name includes the string "sync"; as a result, execution of a high priority process might be slightly delayed.
- The wake_up_locked macro is similar to wake_up, except that it is called when the spin lock in wait_queue_head_t is already held.

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

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

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

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

Process Resource Limits

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

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

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

Table 3-7. Resource limits

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

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

Table 3-7. Resource limits (continued)

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

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

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

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

Process Switch

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

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

Hardware Context

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

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

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

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

- Step-by-step switching performed through a sequence of `mov` instructions allows better control over the validity of the data being loaded. In particular, it is possible to check the values of the `ds` and `es` segmentation registers, which might have been forged by a malicious user. This type of checking is not possible when using a single `far jmp` instruction.
- The amount of time required by the old approach and the new approach is about the same. However, it is not possible to optimize a hardware context switch, while there might be room for improving the current switching code.

Process switching occurs only in Kernel Mode. The contents of all registers used by a process in User Mode have already been saved on the Kernel Mode stack before performing process switching (see Chapter 4). This includes the contents of the `ss` and `esp` pair that specifies the User Mode stack pointer address.

* `far jmp` instructions modify both the `cs` and `eip` registers, while simple `jmp` instructions modify only `eip`.

Task State Segment

The 80×86 architecture includes a specific segment type called the *Task State Segment* (TSS), to store hardware contexts. Although Linux doesn't use hardware context switches, it is nonetheless forced to set up a TSS for each distinct CPU in the system. This is done for two main reasons:

- When an 80×86 CPU switches from User Mode to Kernel Mode, it fetches the address of the Kernel Mode stack from the TSS (see the sections "Hardware Handling of Interrupts and Exceptions" in Chapter 4 and "Issuing a System Call via the sysenter Instruction" in Chapter 10).
- When a User Mode process attempts to access an I/O port by means of an `in` or `out` instruction, the CPU may need to access an I/O Permission Bitmap stored in the TSS to verify whether the process is allowed to address the port.

  More precisely, when a process executes an `in` or `out` I/O instruction in User Mode, the control unit performs the following operations:

  1. It checks the 2-bit IOPL field in the `eflags` register. If it is set to 3, the control unit executes the I/O instructions. Otherwise, it performs the next check.
  2. It accesses the `tr` register to determine the current TSS, and thus the proper I/O Permission Bitmap.
  3. It checks the bit of the I/O Permission Bitmap corresponding to the I/O port specified in the I/O instruction. If it is cleared, the instruction is executed; otherwise, the control unit raises a "General protection" exception.

The `tss_struct` structure describes the format of the TSS. As already mentioned in Chapter 2, the `init_tss` array stores one TSS for each CPU on the system. At each process switch, the kernel updates some fields of the TSS so that the corresponding CPU's control unit may safely retrieve the information it needs. Thus, the TSS reflects the privilege of the current process on the CPU, but there is no need to maintain TSSs for processes when they're not running.

Each TSS has its own 8-byte *Task State Segment Descriptor* (TSSD). This descriptor includes a 32-bit `Base` field that points to the TSS starting address and a 20-bit `Limit` field. The `S` flag of a TSSD is cleared to denote the fact that the corresponding TSS is a System Segment (see the section "Segment Descriptors" in Chapter 2).

The `Type` field is set to either 9 or 11 to denote that the segment is actually a TSS. In the Intel's original design, each process in the system should refer to its own TSS; the second least significant bit of the `Type` field is called the *Busy bit*; it is set to 1 if the process is being executed by a CPU, and to 0 otherwise. In Linux design, there is just one TSS for each CPU, so the Busy bit is always set to 1.

The TSSDs created by Linux are stored in the Global Descriptor Table (GDT), whose base address is stored in the `gdtr` register of each CPU. The `tr` register of each CPU

contains the TSSD Selector of the corresponding TSS. The register also includes two hidden, nonprogrammable fields: the Base and Limit fields of the TSSD. In this way, the processor can address the TSS directly without having to retrieve the TSS address from the GDT.

The thread field

At every process switch, the hardware context of the process being replaced must be saved somewhere. It cannot be saved on the TSS, as in the original Intel design, because Linux uses a single TSS for each processor, instead of one for every process.

Thus, each process descriptor includes a field called thread of type thread_struct, in which the kernel saves the hardware context whenever the process is being switched out. As we'll see later, this data structure includes fields for most of the CPU registers, except the general-purpose registers such as eax, ebx, etc., which are stored in the Kernel Mode stack.

Performing the Process Switch

A process switch may occur at just one well-defined point: the schedule() function, which is discussed at length in Chapter 7. Here, we are only concerned with how the kernel performs a process switch.

Essentially, every process switch consists of two steps:

1. Switching the Page Global Directory to install a new address space; we'll describe this step in Chapter 9.
2. Switching the Kernel Mode stack and the hardware context, which provides all the information needed by the kernel to execute the new process, including the CPU registers.

Again, we assume that prev points to the descriptor of the process being replaced, and next to the descriptor of the process being activated. As we'll see in Chapter 7, prev and next are local variables of the schedule() function.

The switch_to macro

The second step of the process switch is performed by the switch_to macro. It is one of the most hardware-dependent routines of the kernel, and it takes some effort to understand what it does.

First of all, the macro has three parameters, called prev, next, and last. You might easily guess the role of prev and next: they are just placeholders for the local variables prev and next, that is, they are input parameters that specify the memory locations containing the descriptor address of the process being replaced and the descriptor address of the new process, respectively.

What about the third parameter, last? Well, in any process switch three processes are involved, not just two. Suppose the kernel decides to switch off process A and to activate process B. In the schedule() function, prev points to A's descriptor and next points to B's descriptor. As soon as the switch_to macro deactivates A, the execution flow of A freezes.

Later, when the kernel wants to reactivate A, it must switch off another process C (in general, this is different from B) by executing another switch_to macro with prev pointing to C and next pointing to A. When A resumes its execution flow, it finds its old Kernel Mode stack, so the prev local variable points to A's descriptor and next points to B's descriptor. The scheduler, which is now executing on behalf of process A, has lost any reference to C. This reference, however, turns out to be useful to complete the process switching (see Chapter 7 for more details).

The last parameter of the switch_to macro is an output parameter that specifies a memory location in which the macro writes the descriptor address of process C (of course, this is done after A resumes its execution). Before the process switching, the macro saves in the eax CPU register the content of the variable identified by the first input parameter prev—that is, the prev local variable allocated on the Kernel Mode stack of A. After the process switching, when A has resumed its execution, the macro writes the content of the eax CPU register in the memory location of A identified by the third output parameter last. Because the CPU register doesn't change across the process switch, this memory location receives the address of C's descriptor. In the current implementation of schedule(), the last parameter identifies the prev local variable of A, so prev is overwritten with the address of C.

The contents of the Kernel Mode stacks of processes A, B, and C are shown in Figure 3-7, together with the values of the eax register; be warned that the figure shows the value of the prev local variable *before* its value is overwritten with the contents of the eax register.

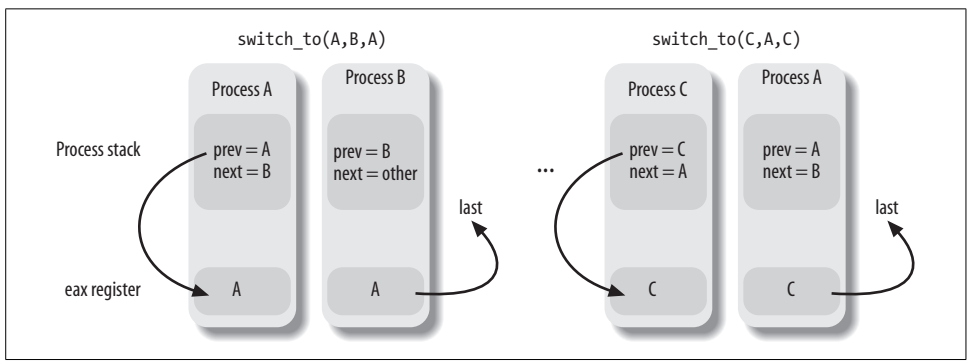


Figure 3-7. Preserving the reference to process C across a process switch

The switch_to macro is coded in *extended inline assembly language* that makes for rather complex reading: in fact, the code refers to registers by means of a special

positional notation that allows the compiler to freely choose the general-purpose registers to be used. Rather than follow the cumbersome extended inline assembly language, we'll describe what the switch_to macro typically does on an 80×86 microprocessor by using standard assembly language:

1. Saves the values of prev and next in the eax and edx registers, respectively:

```
movl prev, %eax
movl next, %edx
```

2. Saves the contents of the eflags and ebp registers in the prev Kernel Mode stack. They must be saved because the compiler assumes that they will stay unchanged until the end of switch_to:

```
pushfl
pushl %ebp
```

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

```
movl %esp,484(%eax)
```

   The 484(%eax) operand identifies the memory cell whose address is the contents of eax plus 484.

4. Loads next->thread.esp in esp. From now on, the kernel operates on the Kernel Mode stack of next, so this instruction performs the actual process switch from prev to next. Because the address of a process descriptor is closely related to that of the Kernel Mode stack (as explained in the section "Identifying a Process" earlier in this chapter), changing the kernel stack means changing the current process:

```
movl 484(%edx), %esp
```

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

```
movl $1f, 480(%eax)
```

6. On the Kernel Mode stack of next, the macro pushes the next->thread.eip value, which, in most cases, is the address labeled as 1:

```
pushl 480(%edx)
```

7. Jumps to the __switch_to( ) C function (see next):

```
jmp __switch_to
```

8. Here process A that was replaced by B gets the CPU again: it executes a few instructions that restore the contents of the eflags and ebp registers. The first of these two instructions is labeled as 1:

```
1:
    popl %ebp
    popfl
```

   Notice how these pop instructions refer to the kernel stack of the prev process. They will be executed when the scheduler selects prev as the new process to be

executed on the CPU, thus invoking `switch_to` with `prev` as the second parameter. Therefore, the `esp` register points to the `prev`'s Kernel Mode stack.

9. Copies the content of the `eax` register (loaded in step 1 above) into the memory location identified by the third parameter `last` of the `switch_to` macro:

   ```
   movl %eax, last
   ```

   As discussed earlier, the `eax` register points to the descriptor of the process that has just been replaced.*

The `__switch_to()` function

The `__switch_to()` function does the bulk of the process switch started by the `switch_to()` macro. It acts on the `prev_p` and `next_p` parameters that denote the former process and the new process. This function call is different from the average function call, though, because `__switch_to()` takes the `prev_p` and `next_p` parameters from the `eax` and `edx` registers (where we saw they were stored), not from the stack like most functions. To force the function to go to the registers for its parameters, the kernel uses the `__attribute__` and `regparm` keywords, which are nonstandard extensions of the C language implemented by the `gcc` compiler. The `__switch_to()` function is declared in the *include/asm-i386/system.h* header file as follows:

```
__switch_to(struct task_struct *prev_p,
            struct task_struct *next_p)
  __attribute__(regparm(3));
```

The steps performed by the function are the following:

1. Executes the code yielded by the `__unlazy_fpu()` macro (see the section "Saving and Loading the FPU, MMX, and XMM Registers" later in this chapter) to optionally save the contents of the FPU, MMX, and XMM registers of the `prev_p` process.

   ```
   __unlazy_fpu(prev_p);
   ```

2. Executes the `smp_processor_id()` macro to get the index of the *local CPU*, namely the CPU that executes the code. The macro gets the index from the `cpu` field of the `thread_info` structure of the current process and stores it into the `cpu` local variable.

3. Loads `next_p->thread.esp0` in the `esp0` field of the TSS relative to the local CPU; as we'll see in the section "Issuing a System Call via the sysenter Instruction" in Chapter 10, any future privilege level change from User Mode to Kernel Mode raised by a `sysenter` assembly instruction will copy this address in the `esp` register:

   ```
   init_tss[cpu].esp0 = next_p->thread.esp0;
   ```

* As stated earlier in this section, the current implementation of the `schedule()` function reuses the `prev` local variable, so that the assembly language instruction looks like `movl %eax,prev`.

4. Loads in the Global Descriptor Table of the local CPU the Thread-Local Storage (TLS) segments used by the next_p process; the three Segment Selectors are stored in the tls_array array inside the process descriptor (see the section "Segmentation in Linux" in Chapter 2).

```
cpu_gdt_table[cpu][6] = next_p->thread.tls_array[0];
cpu_gdt_table[cpu][7] = next_p->thread.tls_array[1];
cpu_gdt_table[cpu][8] = next_p->thread.tls_array[2];
```

5. Stores the contents of the fs and gs segmentation registers in prev_p->thread.fs and prev_p->thread.gs, respectively; the corresponding assembly language instructions are:

```
movl %fs, 40(%esi)
movl %gs, 44(%esi)
```

   The esi register points to the prev_p->thread structure.

6. If the fs or the gs segmentation register have been used either by the prev_p or by the next_p process (i.e., if they have a nonzero value), loads into these registers the values stored in the thread_struct descriptor of the next_p process. This step logically complements the actions performed in the previous step. The main assembly language instructions are:

```
movl 40(%ebx),%fs
movl 44(%ebx),%gs
```

   The ebx register points to the next_p->thread structure. The code is actually more intricate, as an exception might be raised by the CPU when it detects an invalid segment register value. The code takes this possibility into account by adopting a "fix-up" approach (see the section "Dynamic Address Checking: The Fix-up Code" in Chapter 10).

7. Loads six of the dr0, ..., dr7 debug registers* with the contents of the next_p->thread.debugreg array. This is done only if next_p was using the debug registers when it was suspended (that is, field next_p->thread.debugreg[7] is not 0). These registers need not be saved, because the prev_p->thread.debugreg array is modified only when a debugger wants to monitor prev:

```
if (next_p->thread.debugreg[7]){
    loaddebug(&next_p->thread, 0);
    loaddebug(&next_p->thread, 1);
    loaddebug(&next_p->thread, 2);
    loaddebug(&next_p->thread, 3);
    /* no 4 and 5 */
    loaddebug(&next_p->thread, 6);
    loaddebug(&next_p->thread, 7);
}
```

* The 80×86 debug registers allow a process to be monitored by the hardware. Up to four breakpoint areas may be defined. Whenever a monitored process issues a linear address included in one of the breakpoint areas, an exception occurs.

8. Updates the I/O bitmap in the TSS, if necessary. This must be done when either next_p or prev_p has its own customized I/O Permission Bitmap:

```
if (prev_p->thread.io_bitmap_ptr || next_p->thread.io_bitmap_ptr)
    handle_io_bitmap(&next_p->thread, &init_tss[cpu]);
```

   Because processes seldom modify the I/O Permission Bitmap, this bitmap is handled in a "lazy" mode: the actual bitmap is copied into the TSS of the local CPU only if a process actually accesses an I/O port in the current timeslice. The customized I/O Permission Bitmap of a process is stored in a buffer pointed to by the io_bitmap_ptr field of the thread_info structure. The handle_io_bitmap() function sets up the io_bitmap field of the TSS used by the local CPU for the next_p process as follows:

   - If the next_p process does not have its own customized I/O Permission Bitmap, the io_bitmap field of the TSS is set to the value 0x8000.
   - If the next_p process has its own customized I/O Permission Bitmap, the io_bitmap field of the TSS is set to the value 0x9000.

   The io_bitmap field of the TSS should contain an offset inside the TSS where the actual bitmap is stored. The 0x8000 and 0x9000 values point outside of the TSS limit and will thus cause a "General protection" exception whenever the User Mode process attempts to access an I/O port (see the section "Exceptions" in Chapter 4). The do_general_protection() exception handler will check the value stored in the io_bitmap field: if it is 0x8000, the function sends a SIGSEGV signal to the User Mode process; otherwise, if it is 0x9000, the function copies the process bitmap (pointed to by the io_bitmap_ptr field in the thread_info structure) in the TSS of the local CPU, sets the io_bitmap field to the actual bitmap offset (104), and forces a new execution of the faulty assembly language instruction.

9. Terminates. The __switch_to() C function ends by means of the statement:

```
return prev_p;
```

   The corresponding assembly language instructions generated by the compiler are:

```
movl %edi,%eax
ret
```

   The prev_p parameter (now in edi) is copied into eax, because by default the return value of any C function is passed in the eax register. Notice that the value of eax is thus preserved across the invocation of __switch_to(); this is quite important, because the invoking switch_to macro assumes that eax always stores the address of the process descriptor being replaced.

   The ret assembly language instruction loads the eip program counter with the return address stored on top of the stack. However, the __switch_to() function has been invoked simply by jumping into it. Therefore, the ret instruction finds on the stack the address of the instruction labeled as 1, which was pushed by the switch_to macro. If next_p was never suspended before because it is being

executed for the first time, the function finds the starting address of the `ret_from_fork()` function (see the section "The clone(), fork(), and vfork() System Calls" later in this chapter).

Saving and Loading the FPU, MMX, and XMM Registers

Starting with the Intel 80486DX, the arithmetic floating-point unit (FPU) has been integrated into the CPU. The name *mathematical coprocessor* continues to be used in memory of the days when floating-point computations were executed by an expensive special-purpose chip. To maintain compatibility with older models, however, floating-point arithmetic functions are performed with *ESCAPE instructions*, which are instructions with a prefix byte ranging between `0xd8` and `0xdf`. These instructions act on the set of floating-point registers included in the CPU. Clearly, if a process is using ESCAPE instructions, the contents of the floating-point registers belong to its hardware context and should be saved.

In later Pentium models, Intel introduced a new set of assembly language instructions into its microprocessors. They are called *MMX instructions* and are supposed to speed up the execution of multimedia applications. MMX instructions act on the floating-point registers of the FPU. The obvious disadvantage of this architectural choice is that programmers cannot mix floating-point instructions and MMX instructions. The advantage is that operating system designers can ignore the new instruction set, because the same facility of the task-switching code for saving the state of the floating-point unit can also be relied upon to save the MMX state.

MMX instructions speed up multimedia applications, because they introduce a single-instruction multiple-data (SIMD) pipeline inside the processor. The Pentium III model extends that SIMD capability: it introduces the *SSE extensions* (Streaming SIMD Extensions), which adds facilities for handling floating-point values contained in eight 128-bit registers called the XMM registers. Such registers do not overlap with the FPU and MMX registers, so SSE and FPU/MMX instructions may be freely mixed. The Pentium 4 model introduces yet another feature: the *SSE2 extensions*, which is basically an extension of SSE supporting higher-precision floating-point values. SSE2 uses the same set of XMM registers as SSE.

The 80×86 microprocessors do not automatically save the FPU, MMX, and XMM registers in the TSS. However, they include some hardware support that enables kernels to save these registers only when needed. The hardware support consists of a `TS` (Task-Switching) flag in the `cr0` register, which obeys the following rules:

- Every time a hardware context switch is performed, the `TS` flag is set.
- Every time an ESCAPE, MMX, SSE, or SSE2 instruction is executed when the `TS` flag is set, the control unit raises a "Device not available" exception (see Chapter 4).

The TS flag allows the kernel to save and restore the FPU, MMX, and XMM registers only when really needed. To illustrate how it works, suppose that a process A is using the mathematical coprocessor. When a context switch occurs from A to B, the kernel sets the TS flag and saves the floating-point registers into the TSS of process A. If the new process B does not use the mathematical coprocessor, the kernel won't need to restore the contents of the floating-point registers. But as soon as B tries to execute an ESCAPE or MMX instruction, the CPU raises a "Device not available" exception, and the corresponding handler loads the floating-point registers with the values saved in the TSS of process B.

Let's now describe the data structures introduced to handle selective loading of the FPU, MMX, and XMM registers. They are stored in the thread.i387 subfield of the process descriptor, whose format is described by the i387_union union:

```
union i387_union {
    struct i387_fsave_struct    fsave;
    struct i387_fxsave_struct   fxsave;
    struct i387_soft_struct     soft;
};
```

As you see, the field may store just one of three different types of data structures. The i387_soft_struct type is used by CPU models without a mathematical coprocessor; the Linux kernel still supports these old chips by emulating the coprocessor via software. We don't discuss this legacy case further, however. The i387_fsave_struct type is used by CPU models with a mathematical coprocessor and, optionally, an MMX unit. Finally, the i387_fxsave_struct type is used by CPU models featuring SSE and SSE2 extensions.

The process descriptor includes two additional flags:

- The TS_USEDFPU flag, which is included in the status field of the thread_info descriptor. It specifies whether the process used the FPU, MMX, or XMM registers in the current execution run.
- The PF_USED_MATH flag, which is included in the flags field of the task_struct descriptor. This flag specifies whether the contents of the thread.i387 subfield are significant. The flag is cleared (not significant) in two cases, shown in the following list.
  - — When the process starts executing a new program by invoking an execve( ) system call (see Chapter 20). Because control will never return to the former program, the data currently stored in thread.i387 is never used again.
  - — When a process that was executing a program in User Mode starts executing a signal handler procedure (see Chapter 11). Because signal handlers are asynchronous with respect to the program execution flow, the floating-point registers could be meaningless to the signal handler. However, the kernel saves the floating-point registers in thread.i387 before starting the handler and restores them after the handler terminates. Therefore, a signal handler is allowed to use the mathematical coprocessor.

Saving the FPU registers

As stated earlier, the `__switch_to()` function executes the `__unlazy_fpu` macro, passing the process descriptor of the `prev` process being replaced as an argument. The macro checks the value of the `TS_USEDFPU` flags of `prev`. If the flag is set, `prev` has used an FPU, MMX, SSE, or SSE2 instructions; therefore, the kernel must save the relative hardware context:

```
if (prev->thread_info->status & TS_USEDFPU)
    save_init_fpu(prev);
```

The `save_init_fpu()` function, in turn, executes essentially the following operations:

1. Dumps the contents of the FPU registers in the process descriptor of `prev` and then reinitializes the FPU. If the CPU uses SSE/SSE2 extensions, it also dumps the contents of the XMM registers and reinitializes the SSE/SSE2 unit. A couple of powerful extended inline assembly language instructions take care of everything, either:

   ```
   asm volatile( "fxsave %0 ; fnclex"
       : "=m" (prev->thread.i387.fxsave) );
   ```

   if the CPU uses SSE/SSE2 extensions, or otherwise:

   ```
   asm volatile( "fnsave %0 ; fwait"
       : "=m" (prev->thread.i387.fsave) );
   ```

2. Resets the `TS_USEDFPU` flag of `prev`:

   ```
   prev->thread_info->status &= ~TS_USEDFPU;
   ```

3. Sets the TS flag of `cr0` by means of the `stts()` macro, which in practice yields assembly language instructions like the following:

   ```
   movl %cr0, %eax
   orl $8,%eax
   movl %eax, %cr0
   ```

Loading the FPU registers

The contents of the floating-point registers are not restored right after the `next` process resumes execution. However, the `TS` flag of `cr0` has been set by `__unlazy_fpu()`. Thus, the first time the `next` process tries to execute an ESCAPE, MMX, or SSE/SSE2 instruction, the control unit raises a "Device not available" exception, and the kernel (more precisely, the exception handler involved by the exception) runs the `math_state_restore()` function. The `next` process is identified by this handler as `current`.

```
void math_state_restore()
{
    asm volatile ("clts"); /* clear the TS flag of cr0 */
    if (!(current->flags & PF_USED_MATH))
        init_fpu(current);
    restore_fpu(current);
    current->thread.status |= TS_USEDFPU;
}
```

The function clears the TC flags of cr0, so that further FPU, MMX, or SSE/SSE2 instructions executed by the process won't trigger the "Device not available" exception. If the contents of the thread.i387 subfield are not significant, i.e., if the PF_USED_MATH flag is equal to 0, init_fpu() is invoked to reset the thread.i387 subfield and to set the PF_USED_MATH flag of current to 1. The restore_fpu() function is then invoked to load the FPU registers with the proper values stored in the thread.i387 subfield. To do this, either the fxrstor or the frstor assembly language instructions are used, depending on whether the CPU supports SSE/SSE2 extensions. Finally, math_state_restore() sets the TS_USEDFPU flag.

Using the FPU, MMX, and SSE/SSE2 units in Kernel Mode

Even the kernel can make use of the FPU, MMX, or SSE/SSE2 units. In doing so, of course, it should avoid interfering with any computation carried on by the current User Mode process. Therefore:

- Before using the coprocessor, the kernel must invoke kernel_fpu_begin(), which essentially calls save_init_fpu() to save the contents of the registers if the User Mode process used the FPU (TS_USEDFPU flag), and then resets the TS flag of the cr0 register.
- After using the coprocessor, the kernel must invoke kernel_fpu_end(), which sets the TS flag of the cr0 register.

Later, when the User Mode process executes a coprocessor instruction, the math_state_restore() function will restore the contents of the registers, just as in process switch handling.

It should be noted, however, that the execution time of kernel_fpu_begin() is rather large when the current User Mode process is using the coprocessor, so much as to nullify the speedup obtained by using the FPU, MMX, or SSE/SSE2 units. As a matter of fact, the kernel uses them only in a few places, typically when moving or clearing large memory areas or when computing checksum functions.

Creating Processes

Unix operating systems rely heavily on process creation to satisfy user requests. For example, the shell creates a new process that executes another copy of the shell whenever the user enters a command.

Traditional Unix systems treat all processes in the same way: resources owned by the parent process are duplicated in the child process. This approach makes process creation very slow and inefficient, because it requires copying the entire address space of the parent process. The child process rarely needs to read or modify all the resources inherited from the parent; in many cases, it issues an immediate execve() and wipes out the address space that was so carefully copied.

Modern Unix kernels solve this problem by introducing three different mechanisms:

- The Copy On Write technique allows both the parent and the child to read the same physical pages. Whenever either one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process. The implementation of this technique in Linux is fully explained in Chapter 9.
- Lightweight processes allow both the parent and the child to share many per-process kernel data structures, such as the paging tables (and therefore the entire User Mode address space), the open file tables, and the signal dispositions.
- The `vfork( )` system call creates a process that shares the memory address space of its parent. To prevent the parent from overwriting data needed by the child, the parent's execution is blocked until the child exits or executes a new program. We'll learn more about the `vfork( )` system call in the following section.

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

Lightweight processes are created in Linux by using a function named `clone( )`, which uses the following parameters:

`fn`
: Specifies a function to be executed by the new process; when the function returns, the child terminates. The function returns an integer, which represents the exit code for the child process.

`arg`
: Points to data passed to the `fn( )` function.

`flags`
: Miscellaneous information. The low byte specifies the signal number to be sent to the parent process when the child terminates; the `SIGCHLD` signal is generally selected. The remaining three bytes encode a group of clone flags, which are shown in Table 3-8.

`child_stack`
: Specifies the User Mode stack pointer to be assigned to the `esp` register of the child process. The invoking process (the parent) should always allocate a new stack for the child.

`tls`
: Specifies the address of a data structure that defines a Thread Local Storage segment for the new lightweight process (see the section "The Linux GDT" in Chapter 2). Meaningful only if the `CLONE_SETTLS` flag is set.

`ptid`
: Specifies the address of a User Mode variable of the parent process that will hold the PID of the new lightweight process. Meaningful only if the `CLONE_PARENT_SETTID` flag is set.

ctid
: Specifies the address of a User Mode variable of the new lightweight process that will hold the PID of such process. Meaningful only if the CLONE_CHILD_SETTID flag is set.

Table 3-8. Clone flags

Flag name	Description
CLONE_VM	Shares the memory descriptor and all Page Tables (see Chapter 9).
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bitmask used to mask the initial file permissions of a new file (the so-called file umask).
CLONE_FILES	Shares the table that identifies the open files (see Chapter 12).
CLONE_SIGHAND	Shares the tables that identify the signal handlers and the blocked and pending signals (see Chapter 11). If this flag is true, the CLONE_VM flag must also be set.
CLONE_PTRACE	If traced, the parent wants the child to be traced too. Furthermore, the debugger may want to trace the child on its own; in this case, the kernel forces the flag to 1.
CLONE_VFORK	Set when the system call issued is a vfork( ) (see later in this section).
CLONE_PARENT	Sets the parent of the child (parent and real_parent fields in the process descriptor) to the parent of the calling process.
CLONE_THREAD	Inserts the child into the same thread group of the parent, and forces the child to share the signal descriptor of the parent. The child's tgid and group_leader fields are set accordingly. If this flag is true, the CLONE_SIGHAND flag must also be set.
CLONE_NEWNS	Set if the clone needs its own namespace, that is, its own view of the mounted filesystems (see Chapter 12); it is not possible to specify both CLONE_NEWNS and CLONE_FS.
CLONE_SYSVSEM	Shares the System V IPC undoable semaphore operations (see the section "IPC Semaphores" in Chapter 19).
CLONE_SETTLS	Creates a new Thread Local Storage (TLS) segment for the lightweight process; the segment is described in the structure pointed to by the tls parameter.
CLONE_PARENT_SETTID	Writes the PID of the child into the User Mode variable of the parent pointed to by the ptid parameter.
CLONE_CHILD_CLEARTID	When set, the kernel sets up a mechanism to be triggered when the child process will exit or when it will start executing a new program. In these cases, the kernel will clear the User Mode variable pointed to by the ctid parameter and will awaken any process waiting for this event.
CLONE_DETACHED	A legacy flag ignored by the kernel.
CLONE_UNTRACED	Set by the kernel to override the value of the CLONE_PTRACE flag (used for disabling tracing of kernel threads; see the section "Kernel Threads" later in this chapter).
CLONE_CHILD_SETTID	Writes the PID of the child into the User Mode variable of the child pointed to by the ctid parameter.
CLONE_STOPPED	Forces the child to start in the TASK_STOPPED state.

clone( ) is actually a wrapper function defined in the C library (see the section "POSIX APIs and System Calls" in Chapter 10), which sets up the stack of the new

lightweight process and invokes a clone() system call hidden to the programmer. The sys_clone() service routine that implements the clone() system call does not have the fn and arg parameters. In fact, the wrapper function saves the pointer fn into the child's stack position corresponding to the return address of the wrapper function itself; the pointer arg is saved on the child's stack right below fn. When the wrapper function terminates, the CPU fetches the return address from the stack and executes the fn(arg) function.

The traditional fork() system call is implemented by Linux as a clone() system call whose flags parameter specifies both a SIGCHLD signal and all the clone flags cleared, and whose child_stack parameter is the current parent stack pointer. Therefore, the parent and child temporarily share the same User Mode stack. But thanks to the Copy On Write mechanism, they usually get separate copies of the User Mode stack as soon as one tries to change the stack.

The vfork() system call, introduced in the previous section, is implemented by Linux as a clone() system call whose flags parameter specifies both a SIGCHLD signal and the flags CLONE_VM and CLONE_VFORK, and whose child_stack parameter is equal to the current parent stack pointer.

The do_fork() function

The do_fork() function, which handles the clone(), fork(), and vfork() system calls, acts on the following parameters:

clone_flags
: Same as the flags parameter of clone()

stack_start
: Same as the child_stack parameter of clone()

regs
: Pointer to the values of the general purpose registers saved into the Kernel Mode stack when switching from User Mode to Kernel Mode (see the section "The do_IRQ() function" in Chapter 4)

stack_size
: Unused (always set to 0)

parent_tidptr, child_tidptr
: Same as the corresponding ptid and ctid parameters of clone()

do_fork() makes use of an auxiliary function called copy_process() to set up the process descriptor and any other kernel data structure required for child's execution. Here are the main steps performed by do_fork():

1. Allocates a new PID for the child by looking in the pidmap_array bitmap (see the earlier section "Identifying a Process").
2. Checks the ptrace field of the parent (current->ptrace): if it is not zero, the parent process is being traced by another process, thus do_fork() checks whether

the debugger wants to trace the child on its own (independently of the value of the CLONE_PTRACE flag specified by the parent); in this case, if the child is not a kernel thread (CLONE_UNTRACED flag cleared), the function sets the CLONE_PTRACE flag.

3. Invokes copy_process() to make a copy of the process descriptor. If all needed resources are available, this function returns the address of the task_struct descriptor just created. This is the workhorse of the forking procedure, and we will describe it right after do_fork().
4. If either the CLONE_STOPPED flag is set or the child process must be traced, that is, the PT_PTRACED flag is set in p->ptrace, it sets the state of the child to TASK_STOPPED and adds a pending SIGSTOP signal to it (see the section "The Role of Signals" in Chapter 11). The state of the child will remain TASK_STOPPED until another process (presumably the tracing process or the parent) will revert its state to TASK_RUNNING, usually by means of a SIGCONT signal.
5. If the CLONE_STOPPED flag is not set, it invokes the wake_up_new_task() function, which performs the following operations:
   a. Adjusts the scheduling parameters of both the parent and the child (see "The Scheduling Algorithm" in Chapter 7).
   b. If the child will run on the same CPU as the parent,* and parent and child do not share the same set of page tables (CLONE_VM flag cleared), it then forces the child to run before the parent by inserting it into the parent's runqueue right before the parent. This simple step yields better performance if the child flushes its address space and executes a new program right after the forking. If we let the parent run first, the Copy On Write mechanism would give rise to a series of unnecessary page duplications.
   c. Otherwise, if the child will not be run on the same CPU as the parent, or if parent and child share the same set of page tables (CLONE_VM flag set), it inserts the child in the last position of the parent's runqueue.
6. If the CLONE_STOPPED flag is set, it puts the child in the TASK_STOPPED state.
7. If the parent process is being traced, it stores the PID of the child in the ptrace_message field of current and invokes ptrace_notify(), which essentially stops the current process and sends a SIGCHLD signal to its parent. The "grandparent" of the child is the debugger that is tracing the parent; the SIGCHLD signal notifies the debugger that current has forked a child, whose PID can be retrieved by looking into the current->ptrace_message field.
8. If the CLONE_VFORK flag is specified, it inserts the parent process in a wait queue and suspends it until the child releases its memory address space (that is, until the child either terminates or executes a new program).
9. Terminates by returning the PID of the child.

* The parent process might be moved on to another CPU while the kernel forks the new process.

The copy_process() function

The copy_process() function sets up the process descriptor and any other kernel data structure required for a child's execution. Its parameters are the same as do_fork(), plus the PID of the child. Here is a description of its most significant steps:

1. Checks whether the flags passed in the clone_flags parameter are compatible. In particular, it returns an error code in the following cases:
   a. Both the flags CLONE_NEWNS and CLONE_FS are set.
   b. The CLONE_THREAD flag is set, but the CLONE_SIGHAND flag is cleared (lightweight processes in the same thread group must share signals).
   c. The CLONE_SIGHAND flag is set, but the CLONE_VM flag is cleared (lightweight processes sharing the signal handlers must also share the memory descriptor).
2. Performs any additional security checks by invoking security_task_create() and, later, security_task_alloc(). The Linux kernel 2.6 offers hooks for security extensions that enforce a security model stronger than the one adopted by traditional Unix. See Chapter 20 for details.
3. Invokes dup_task_struct() to get the process descriptor for the child. This function performs the following actions:
   a. Invokes __unlazy_fpu() on the current process to save, if necessary, the contents of the FPU, MMX, and SSE/SSE2 registers in the thread_info structure of the parent. Later, dup_task_struct() will copy these values in the thread_info structure of the child.
   b. Executes the alloc_task_struct() macro to get a process descriptor (task_struct structure) for the new process, and stores its address in the tsk local variable.
   c. Executes the alloc_thread_info macro to get a free memory area to store the thread_info structure and the Kernel Mode stack of the new process, and saves its address in the ti local variable. As explained in the earlier section "Identifying a Process," the size of this memory area is either 8 KB or 4 KB.
   d. Copies the contents of the current's process descriptor into the task_struct structure pointed to by tsk, then sets tsk->thread_info to ti.
   e. Copies the contents of the current's thread_info descriptor into the structure pointed to by ti, then sets ti->task to tsk.
   f. Sets the usage counter of the new process descriptor (tsk->usage) to 2 to specify that the process descriptor is in use and that the corresponding process is alive (its state is not EXIT_ZOMBIE or EXIT_DEAD).
   g. Returns the process descriptor pointer of the new process (tsk).
4. Checks whether the value stored in current->signal->rlim[RLIMIT_NPROC].rlim_cur is smaller than or equal to the current number of processes owned by the user. If so, an error code is returned, unless the process has root privileges. The

function gets the current number of processes owned by the user from a per-user data structure named user_struct. This data structure can be found through a pointer in the user field of the process descriptor.

5. Increases the usage counter of the user_struct structure (tsk->user->__count field) and the counter of the processes owned by the user (tsk->user->processes).
6. Checks that the number of processes in the system (stored in the nr_threads variable) does not exceed the value of the max_threads variable. The default value of this variable depends on the amount of RAM in the system. The general rule is that the space taken by all thread_info descriptors and Kernel Mode stacks cannot exceed 1/8 of the physical memory. However, the system administrator may change this value by writing in the */proc/sys/kernel/threads-max* file.
7. If the kernel functions implementing the execution domain and the executable format (see Chapter 20) of the new process are included in kernel modules, it increases their usage counters (see Appendix B).
8. Sets a few crucial fields related to the process state:
   a. Initializes the big kernel lock counter tsk->lock_depth to -1 (see the section "The Big Kernel Lock" in Chapter 5).
   b. Initializes the tsk->did_exec field to 0: it counts the number of execve() system calls issued by the process.
   c. Updates some of the flags included in the tsk->flags field that have been copied from the parent process: first clears the PF_SUPERPRIV flag, which indicates whether the process has used any of its superuser privileges, then sets the PF_FORKNOEXEC flag, which indicates that the child has not yet issued an execve() system call.
9. Stores the PID of the new process in the tsk->pid field.
10. If the CLONE_PARENT_SETTID flag in the clone_flags parameter is set, it copies the child's PID into the User Mode variable addressed by the parent_tidptr parameter.
11. Initializes the list_head data structures and the spin locks included in the child's process descriptor, and sets up several other fields related to pending signals, timers, and time statistics.
12. Invokes copy_semundo(), copy_files(), copy_fs(), copy_sighand(), copy_signal(), copy_mm(), and copy_namespace() to create new data structures and copy into them the values of the corresponding parent process data structures, unless specified differently by the clone_flags parameter.
13. Invokes copy_thread() to initialize the Kernel Mode stack of the child process with the values contained in the CPU registers when the clone() system call was issued (these values have been saved in the Kernel Mode stack of the parent, as described in Chapter 10). However, the function forces the value 0 into the field

corresponding to the eax register (this is the child's return value of the fork() or clone() system call). The thread.esp field in the descriptor of the child process is initialized with the base address of the child's Kernel Mode stack, and the address of an assembly language function (ret_from_fork()) is stored in the thread.eip field. If the parent process makes use of an I/O Permission Bitmap, the child gets a copy of such bitmap. Finally, if the CLONE_SETTLS flag is set, the child gets the TLS segment specified by the User Mode data structure pointed to by the tls parameter of the clone() system call.*

14. If either CLONE_CHILD_SETTID or CLONE_CHILD_CLEARTID is set in the clone_flags parameter, it copies the value of the child_tidptr parameter in the tsk->set_chid_tid or tsk->clear_child_tid field, respectively. These flags specify that the value of the variable pointed to by child_tidptr in the User Mode address space of the child has to be changed, although the actual write operations will be done later.
15. Turns off the TIF_SYSCALL_TRACE flag in the thread_info structure of the child, so that the ret_from_fork() function will not notify the debugging process about the system call termination (see the section "Entering and Exiting a System Call" in Chapter 10). (The system call tracing of the child is not disabled, because it is controlled by the PTRACE_SYSCALL flag in tsk->ptrace.)
16. Initializes the tsk->exit_signal field with the signal number encoded in the low bits of the clone_flags parameter, unless the CLONE_THREAD flag is set, in which case initializes the field to -1. As we'll see in the section "Process Termination" later in this chapter, only the death of the last member of a thread group (usually, the thread group leader) causes a signal notifying the parent of the thread group leader.
17. Invokes sched_fork() to complete the initialization of the scheduler data structure of the new process. The function also sets the state of the new process to TASK_RUNNING and sets the preempt_count field of the thread_info structure to 1, thus disabling kernel preemption (see the section "Kernel Preemption" in Chapter 5). Moreover, in order to keep process scheduling fair, the function shares the remaining timeslice of the parent between the parent and the child (see "The scheduler_tick() Function" in Chapter 7).
18. Sets the cpu field in the thread_info structure of the new process to the number of the local CPU returned by smp_processor_id().
19. Initializes the fields that specify the parenthood relationships. In particular, if CLONE_PARENT or CLONE_THREAD are set, it initializes tsk->real_parent and tsk->

* A careful reader might wonder how copy_thread() gets the value of the tls parameter of clone(), because tls is not passed to do_fork() and nested functions. As we'll see in Chapter 10, the parameters of the system calls are usually passed to the kernel by copying their values into some CPU register; thus, these values are saved in the Kernel Mode stack together with the other registers. The copy_thread() function just looks at the address saved in the Kernel Mode stack location corresponding to the value of esi.

parent to the value in current->real_parent; the parent of the child thus appears as the parent of the current process. Otherwise, it sets the same fields to current.

20. If the child does not need to be traced (CLONE_PTRACE flag not set), it sets the tsk->ptrace field to 0. This field stores a few flags used when a process is being traced by another process. In such a way, even if the current process is being traced, the child will not.
21. Executes the SET_LINKS macro to insert the new process descriptor in the process list.
22. If the child must be traced (PT_PTRACED flag in the tsk->ptrace field set), it sets tsk->parent to current->parent and inserts the child into the trace list of the debugger.
23. Invokes attach_pid( ) to insert the PID of the new process descriptor in the pidhash[PIDTYPE_PID] hash table.
24. If the child is a thread group leader (flag CLONE_THREAD cleared):
    a. Initializes tsk->tgid to tsk->pid.
    b. Initializes tsk->group_leader to tsk.
    c. Invokes three times attach_pid( ) to insert the child in the PID hash tables of type PIDTYPE_TGID, PIDTYPE_PGID, and PIDTYPE_SID.
25. Otherwise, if the child belongs to the thread group of its parent (CLONE_THREAD flag set):
    a. Initializes tsk->tgid to tsk->current->tgid.
    b. Initializes tsk->group_leader to the value in current->group_leader.
    c. Invokes attach_pid( ) to insert the child in the PIDTYPE_TGID hash table (more specifically, in the per-PID list of the current->group_leader process).
26. A new process has now been added to the set of processes: increases the value of the nr_threads variable.
27. Increases the total_forks variable to keep track of the number of forked processes.
28. Terminates by returning the child's process descriptor pointer (tsk).

Let's go back to what happens after do_fork( ) terminates. Now we have a complete child process in the runnable state. But it isn't actually running. It is up to the scheduler to decide when to give the CPU to this child. At some future process switch, the schedule bestows this favor on the child process by loading a few CPU registers with the values of the thread field of the child's process descriptor. In particular, esp is loaded with thread.esp (that is, with the address of child's Kernel Mode stack), and eip is loaded with the address of ret_from_fork( ). This assembly language function invokes the schedule_tail( ) function (which in turn invokes the finish_task_switch( ) function to complete the process switch; see the section "The schedule( ) Function" in Chapter 7), reloads all other registers with the values stored in the

stack, and forces the CPU back to User Mode. The new process then starts its execution right at the end of the fork(), vfork(), or clone() system call. The value returned by the system call is contained in eax: the value is 0 for the child and equal to the PID for the child's parent. To understand how this is done, look back at what copy_thread() does on the eax register of the child's process (step 13 of copy_process()).

The child process executes the same code as the parent, except that the fork returns a 0 (see step 13 of copy_process()). The developer of the application can exploit this fact, in a manner familiar to Unix programmers, by inserting a conditional statement in the program based on the PID value that forces the child to behave differently from the parent process.

Kernel Threads

Traditional Unix systems delegate some critical tasks to intermittently running processes, including flushing disk caches, swapping out unused pages, servicing network connections, and so on. Indeed, it is not efficient to perform these tasks in strict linear fashion; both their functions and the end user processes get better response if they are scheduled in the background. Because some of the system processes run only in Kernel Mode, modern operating systems delegate their functions to *kernel threads*, which are not encumbered with the unnecessary User Mode context. In Linux, kernel threads differ from regular processes in the following ways:

- Kernel threads run only in Kernel Mode, while regular processes run alternatively in Kernel Mode and in User Mode.
- Because kernel threads run only in Kernel Mode, they use only linear addresses greater than PAGE_OFFSET. Regular processes, on the other hand, use all four gigabytes of linear addresses, in either User Mode or Kernel Mode.

Creating a kernel thread

The kernel_thread() function creates a new kernel thread. It receives as parameters the address of the kernel function to be executed (fn), the argument to be passed to that function (arg), and a set of clone flags (flags). The function essentially invokes do_fork() as follows:

```
do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, pregs, 0, NULL, NULL);
```

The CLONE_VM flag avoids the duplication of the page tables of the calling process: this duplication would be a waste of time and memory, because the new kernel thread will not access the User Mode address space anyway. The CLONE_UNTRACED flag ensures that no process will be able to trace the new kernel thread, even if the calling process is being traced.

The pregs parameter passed to do_fork() corresponds to the address in the Kernel Mode stack where the copy_thread() function will find the initial values of the CPU

registers for the new thread. The kernel_thread() function builds up this stack area so that:

- The ebx and edx registers will be set by copy_thread() to the values of the parameters fn and arg, respectively.
- The eip register will be set to the address of the following assembly language fragment:

```
movl %edx,%eax
pushl %edx
call *%ebx
pushl %eax
call do_exit
```

Therefore, the new kernel thread starts by executing the fn(arg) function. If this function terminates, the kernel thread executes the _exit() system call passing to it the return value of fn() (see the section "Destroying Processes" later in this chapter).

Process 0

The ancestor of all processes, called *process 0*, the *idle process,* or, for historical reasons, the *swapper process*, is a kernel thread created from scratch during the initialization phase of Linux (see Appendix A). This ancestor process uses the following statically allocated data structures (data structures for all other processes are dynamically allocated):

- A process descriptor stored in the init_task variable, which is initialized by the INIT_TASK macro.
- A thread_info descriptor and a Kernel Mode stack stored in the init_thread_union variable and initialized by the INIT_THREAD_INFO macro.
- The following tables, which the process descriptor points to:
  - init_mm
  - init_fs
  - init_files
  - init_signals
  - init_sighand

  The tables are initialized, respectively, by the following macros:
  - INIT_MM
  - INIT_FS
  - INIT_FILES
  - INIT_SIGNALS
  - INIT_SIGHAND

- The master kernel Page Global Directory stored in `swapper_pg_dir` (see the section "Kernel Page Tables" in Chapter 2).

The `start_kernel()` function initializes all the data structures needed by the kernel, enables interrupts, and creates another kernel thread, named *process 1* (more commonly referred to as the *init process*):

```
kernel_thread(init, NULL, CLONE_FS|CLONE_SIGHAND);
```

The newly created kernel thread has PID 1 and shares all per-process kernel data structures with process 0. When selected by the scheduler, the *init* process starts executing the `init()` function.

After having created the *init* process, process 0 executes the `cpu_idle()` function, which essentially consists of repeatedly executing the `hlt` assembly language instruction with the interrupts enabled (see Chapter 4). Process 0 is selected by the scheduler only when there are no other processes in the `TASK_RUNNING` state.

In multiprocessor systems there is a process 0 for each CPU. Right after the power-on, the BIOS of the computer starts a single CPU while disabling the others. The swapper process running on CPU 0 initializes the kernel data structures, then enables the other CPUs and creates the additional *swapper* processes by means of the `copy_process()` function passing to it the value 0 as the new PID. Moreover, the kernel sets the `cpu` field of the `thread_info` descriptor of each forked process to the proper CPU index.

Process 1

The kernel thread created by process 0 executes the `init()` function, which in turn completes the initialization of the kernel. Then `init()` invokes the `execve()` system call to load the executable program *init*. As a result, the *init* kernel thread becomes a regular process having its own per-process kernel data structure (see Chapter 20). The *init* process stays alive until the system is shut down, because it creates and monitors the activity of all processes that implement the outer layers of the operating system.

Other kernel threads

Linux uses many other kernel threads. Some of them are created in the initialization phase and run until shutdown; others are created "on demand," when the kernel must execute a task that is better performed in its own execution context.

A few examples of kernel threads (besides process 0 and process 1) are:

keventd (also called events)
: Executes the functions in the `keventd_wq` workqueue (see Chapter 4).

kapmd
: Handles the events related to the Advanced Power Management (APM).

kswapd
: Reclaims memory, as described in the section "Periodic Reclaiming" in Chapter 17.

pdflush
: Flushes "dirty" buffers to disk to reclaim memory, as described in the section "The pdflush Kernel Threads" in Chapter 15.

kblockd
: Executes the functions in the `kblockd_workqueue` workqueue. Essentially, it periodically activates the block device drivers, as described in the section "Activating the Block Device Driver" in Chapter 14.

ksoftirqd
: Runs the tasklets (see section "Softirqs and Tasklets" in Chapter 4); there is one of these kernel threads for each CPU in the system.

Destroying Processes

Most processes "die" in the sense that they terminate the execution of the code they were supposed to run. When this occurs, the kernel must be notified so that it can release the resources owned by the process; this includes memory, open files, and any other odds and ends that we will encounter in this book, such as semaphores.

The usual way for a process to terminate is to invoke the `exit()` library function, which releases the resources allocated by the C library, executes each function registered by the programmer, and ends up invoking a system call that evicts the process from the system. The `exit()` library function may be inserted by the programmer explicitly. Additionally, the C compiler always inserts an `exit()` function call right after the last statement of the `main()` function.

Alternatively, the kernel may force a whole thread group to die. This typically occurs when a process in the group has received a signal that it cannot handle or ignore (see Chapter 11) or when an unrecoverable CPU exception has been raised in Kernel Mode while the kernel was running on behalf of the process (see Chapter 4).

Process Termination

In Linux 2.6 there are two system calls that terminate a User Mode application:

- The `exit_group()` system call, which terminates a full thread group, that is, a whole multithreaded application. The main kernel function that implements this system call is called `do_group_exit()`. This is the system call that should be invoked by the `exit()` C library function.
- The `_exit()` system call, which terminates a single process, regardless of any other process in the thread group of the victim. The main kernel function that

implements this system call is called do_exit(). This is the system call invoked, for instance, by the pthread_exit() function of the LinuxThreads library.

The do_group_exit() function

The do_group_exit() function kills all processes belonging to the thread group of current. It receives as a parameter the process termination code, which is either a value specified in the exit_group() system call (normal termination) or an error code supplied by the kernel (abnormal termination). The function executes the following operations:

1. Checks whether the SIGNAL_GROUP_EXIT flag of the exiting process is not zero, which means that the kernel already started an exit procedure for this thread group. In this case, it considers as exit code the value stored in current->signal->group_exit_code, and jumps to step 4.
2. Otherwise, it sets the SIGNAL_GROUP_EXIT flag of the process and stores the termination code in the current->signal->group_exit_code field.
3. Invokes the zap_other_threads() function to kill the other processes in the thread group of current, if any. In order to do this, the function scans the per-PID list in the PIDTYPE_TGID hash table corresponding to current->tgid; for each process in the list different from current, it sends a SIGKILL signal to it (see Chapter 11). As a result, all such processes will eventually execute the do_exit() function, and thus they will be killed.
4. Invokes the do_exit() function passing to it the process termination code. As we'll see below, do_exit() kills the process and never returns.

The do_exit() function

All process terminations are handled by the do_exit() function, which removes most references to the terminating process from kernel data structures. The do_exit() function receives as a parameter the process termination code and essentially executes the following actions:

1. Sets the PF_EXITING flag in the flag field of the process descriptor to indicate that the process is being eliminated.
2. Removes, if necessary, the process descriptor from a dynamic timer queue via the del_timer_sync() function (see Chapter 6).
3. Detaches from the process descriptor the data structures related to paging, semaphores, filesystem, open file descriptors, namespaces, and I/O Permission Bitmap, respectively, with the exit_mm(), exit_sem(), __exit_files(), __exit_fs(), exit_namespace(), and exit_thread() functions. These functions also remove each of these data structures if no other processes are sharing them.

4. If the kernel functions implementing the execution domain and the executable format (see Chapter 20) of the process being killed are included in kernel modules, the function decreases their usage counters.
5. Sets the `exit_code` field of the process descriptor to the process termination code. This value is either the `_exit()` or `exit_group()` system call parameter (normal termination), or an error code supplied by the kernel (abnormal termination).
6. Invokes the `exit_notify()` function to perform the following operations:
   a. Updates the parenthood relationships of both the parent process and the child processes. All child processes created by the terminating process become children of another process in the same thread group, if any is running, or otherwise of the *init* process.
   b. Checks whether the `exit_signal` process descriptor field of the process being terminated is different from -1, and whether the process is the last member of its thread group (notice that these conditions always hold for any normal process; see step 16 in the description of `copy_process()` in the earlier section "The clone(), fork(), and vfork() System Calls"). In this case, the function sends a signal (usually `SIGCHLD`) to the parent of the process being terminated to notify the parent about a child's death.
   c. Otherwise, if the `exit_signal` field is equal to -1 or the thread group includes other processes, the function sends a `SIGCHLD` signal to the parent only if the process is being traced (in this case the parent is the debugger, which is thus informed of the death of the lightweight process).
   d. If the `exit_signal` process descriptor field is equal to -1 and the process is not being traced, it sets the `exit_state` field of the process descriptor to `EXIT_DEAD`, and invokes `release_task()` to reclaim the memory of the remaining process data structures and to decrease the usage counter of the process descriptor (see the following section). The usage counter becomes equal to 1 (see step 3f in the `copy_process()` function), so that the process descriptor itself is not released right away.
   e. Otherwise, if the `exit_signal` process descriptor field is not equal to -1 or the process is being traced, it sets the `exit_state` field to `EXIT_ZOMBIE`. We'll see what happens to zombie processes in the following section.
   f. Sets the `PF_DEAD` flag in the `flags` field of the process descriptor (see the section "The schedule() Function" in Chapter 7).
7. Invokes the `schedule()` function (see Chapter 7) to select a new process to run. Because a process in an `EXIT_ZOMBIE` state is ignored by the scheduler, the process stops executing right after the `switch_to` macro in `schedule()` is invoked. As we'll see in Chapter 7, the scheduler will check the `PF_DEAD` flag and will decrease

the usage counter in the descriptor of the zombie process being replaced to denote the fact that the process is no longer alive.

Process Removal

The Unix operating system allows a process to query the kernel to obtain the PID of its parent process or the execution state of any of its children. A process may, for instance, create a child process to perform a specific task and then invoke some `wait()`-like library function to check whether the child has terminated. If the child has terminated, its termination code will tell the parent process if the task has been carried out successfully.

To comply with these design choices, Unix kernels are not allowed to discard data included in a process descriptor field right after the process terminates. They are allowed to do so only after the parent process has issued a `wait()`-like system call that refers to the terminated process. This is why the `EXIT_ZOMBIE` state has been introduced: although the process is technically dead, its descriptor must be saved until the parent process is notified.

What happens if parent processes terminate before their children? In such a case, the system could be flooded with zombie processes whose process descriptors would stay forever in RAM. As mentioned earlier, this problem is solved by forcing all orphan processes to become children of the *init* process. In this way, the *init* process will destroy the zombies while checking for the termination of one of its legitimate children through a `wait()`-like system call.

The `release_task()` function detaches the last data structures from the descriptor of a zombie process; it is applied on a zombie process in two possible ways: by the `do_exit()` function if the parent is not interested in receiving signals from the child, or by the `wait4()` or `waitpid()` system calls after a signal has been sent to the parent. In the latter case, the function also will reclaim the memory used by the process descriptor, while in the former case the memory reclaiming will be done by the scheduler (see Chapter 7). This function executes the following steps:

1. Decreases the number of processes belonging to the user owner of the terminated process. This value is stored in the `user_struct` structure mentioned earlier in the chapter (see step 4 of `copy_process()`).
2. If the process is being traced, the function removes it from the debugger's `ptrace_children` list and assigns the process back to its original parent.
3. Invokes `__exit_signal()` to cancel any pending signal and to release the `signal_struct` descriptor of the process. If the descriptor is no longer used by other lightweight processes, the function also removes this data structure. Moreover, the function invokes `exit_itimers()` to detach any POSIX interval timer from the process.

4. Invokes `__exit_sighand()` to get rid of the signal handlers.
5. Invokes `__unhash_process()`, which in turn:
   a. Decreases by 1 the `nr_threads` variable.
   b. Invokes `detach_pid()` twice to remove the process descriptor from the `pidhash` hash tables of type `PIDTYPE_PID` and `PIDTYPE_TGID`.
   c. If the process is a thread group leader, invokes again `detach_pid()` twice to remove the process descriptor from the `PIDTYPE_PGID` and `PIDTYPE_SID` hash tables.
   d. Uses the `REMOVE_LINKS` macro to unlink the process descriptor from the process list.
6. If the process is not a thread group leader, the leader is a zombie, and the process is the last member of the thread group, the function sends a signal to the parent of the leader to notify it of the death of the process.
7. Invokes the `sched_exit()` function to adjust the timeslice of the parent process (this step logically complements step 17 in the description of `copy_process()`)
8. Invokes `put_task_struct()` to decrease the process descriptor's usage counter; if the counter becomes zero, the function drops any remaining reference to the process:
   a. Decreases the usage counter (`__count` field) of the `user_struct` data structure of the user that owns the process (see step 5 of `copy_process()`), and releases that data structure if the usage counter becomes zero.
   b. Releases the process descriptor and the memory area used to contain the `thread_info` descriptor and the Kernel Mode stack.

CHAPTER 5

Kernel Synchronization

You could think of the kernel as a server that answers requests; these requests can come either from a process running on a CPU or an external device issuing an interrupt request. We make this analogy to underscore that parts of the kernel are not run serially, but in an interleaved way. Thus, they can give rise to race conditions, which must be controlled through proper synchronization techniques. A general introduction to these topics can be found in the section "An Overview of Unix Kernels" in Chapter 1.

We start this chapter by reviewing when, and to what extent, kernel requests are executed in an interleaved fashion. We then introduce the basic synchronization primitives implemented by the kernel and describe how they are applied in the most common cases. Finally, we illustrate a few practical examples.

How the Kernel Services Requests

To get a better grasp of how kernel's code is executed, we will look at the kernel as a waiter who must satisfy two types of requests: those issued by customers and those issued by a limited number of different bosses. The policy adopted by the waiter is the following:

1. If a boss calls while the waiter is idle, the waiter starts servicing the boss.
2. If a boss calls while the waiter is servicing a customer, the waiter stops servicing the customer and starts servicing the boss.
3. If a boss calls while the waiter is servicing another boss, the waiter stops servicing the first boss and starts servicing the second one. When he finishes servicing the new boss, he resumes servicing the former one.
4. One of the bosses may induce the waiter to leave the customer being currently serviced. After servicing the last request of the bosses, the waiter may decide to drop temporarily his customer and to pick up a new one.

The services performed by the waiter correspond to the code executed when the CPU is in Kernel Mode. If the CPU is executing in User Mode, the waiter is considered idle.

Boss requests correspond to interrupts, while customer requests correspond to system calls or exceptions raised by User Mode processes. As we shall see in detail in Chapter 10, User Mode processes that want to request a service from the kernel must issue an appropriate instruction (on the 80×86, an `int $0x80` or a `sysenter` instruction). Such instructions raise an exception that forces the CPU to switch from User Mode to Kernel Mode. In the rest of this chapter, we will generally denote as "exceptions" both the system calls and the usual exceptions.

The careful reader has already associated the first three rules with the nesting of kernel control paths described in "Nested Execution of Exception and Interrupt Handlers" in Chapter 4. The fourth rule corresponds to one of the most interesting new features included in the Linux 2.6 kernel, namely *kernel preemption*.

Kernel Preemption

It is surprisingly hard to give a good definition of kernel preemption. As a first try, we could say that a kernel is *preemptive* if a process switch may occur while the replaced process is executing a kernel function, that is, while it runs in Kernel Mode. Unfortunately, in Linux (as well as in any other real operating system) things are much more complicated:

- Both in preemptive and nonpreemptive kernels, a process running in Kernel Mode can voluntarily relinquish the CPU, for instance because it has to sleep waiting for some resource. We will call this kind of process switch a *planned process switch*. However, a preemptive kernel differs from a nonpreemptive kernel on the way a process running in Kernel Mode reacts to asynchronous events that could induce a process switch—for instance, an interrupt handler that awakes a higher priority process. We will call this kind of process switch a *forced process switch*.
- All process switches are performed by the `switch_to` macro. In both preemptive and nonpreemptive kernels, a process switch occurs when a process has finished some thread of kernel activity and the scheduler is invoked. However, in nonpreemptive kernels, the current process cannot be replaced unless it is about to switch to User Mode (see the section "Performing the Process Switch" in Chapter 3).

Therefore, the main characteristic of a preemptive kernel is that a process running in Kernel Mode can be replaced by another process while in the middle of a kernel function.

Let's give a couple of examples to illustrate the difference between preemptive and nonpreemptive kernels.

While process A executes an exception handler (necessarily in Kernel Mode), a higher priority process B becomes runnable. This could happen, for instance, if an IRQ occurs

and the corresponding handler awakens process B. If the kernel is preemptive, a forced process switch replaces process A with B. The exception handler is left unfinished and will be resumed only when the scheduler selects again process A for execution. Conversely, if the kernel is nonpreemptive, no process switch occurs until process A either finishes handling the exception handler or voluntarily relinquishes the CPU.

For another example, consider a process that executes an exception handler and whose time quantum expires (see the section "The scheduler_tick() Function" in Chapter 7). If the kernel is preemptive, the process may be replaced immediately; however, if the kernel is nonpreemptive, the process continues to run until it finishes handling the exception handler or voluntarily relinquishes the CPU.

The main motivation for making a kernel preemptive is to reduce the *dispatch latency* of the User Mode processes, that is, the delay between the time they become runnable and the time they actually begin running. Processes performing timely scheduled tasks (such as external hardware controllers, environmental monitors, movie players, and so on) really benefit from kernel preemption, because it reduces the risk of being delayed by another process running in Kernel Mode.

Making the Linux 2.6 kernel preemptive did not require a drastic change in the kernel design with respect to the older nonpreemptive kernel versions. As described in the section "Returning from Interrupts and Exceptions" in Chapter 4, kernel preemption is disabled when the `preempt_count` field in the `thread_info` descriptor referenced by the `current_thread_info()` macro is greater than zero. The field encodes three different counters, as shown in Table 4-10 in Chapter 4, so it is greater than zero when any of the following cases occurs:

1. The kernel is executing an interrupt service routine.
2. The deferrable functions are disabled (always true when the kernel is executing a softirq or tasklet).
3. The kernel preemption has been explicitly disabled by setting the preemption counter to a positive value.

The above rules tell us that the kernel can be preempted only when it is executing an exception handler (in particular a system call) and the kernel preemption has not been explicitly disabled. Furthermore, as described in the section "Returning from Interrupts and Exceptions" in Chapter 4, the local CPU must have local interrupts enabled, otherwise kernel preemption is not performed.

A few simple macros listed in Table 5-1 deal with the preemption counter in the `prempt_count` field.

Table 5-1. Macros dealing with the preemption counter subfield

Macro	Description
`preempt_count()`	Selects the `preempt_count` field in the `thread_info` descriptor
`preempt_disable()`	Increases by one the value of the preemption counter

Table 5-1. Macros dealing with the preemption counter subfield (continued)

Macro	Description
`preempt_enable_no_resched()`	Decreases by one the value of the preemption counter
`preempt_enable()`	Decreases by one the value of the preemption counter, and invokes `preempt_schedule()` if the `TIF_NEED_RESCHED` flag in the `thread_info` descriptor is set
`get_cpu()`	Similar to `preempt_disable()`, but also returns the number of the local CPU
`put_cpu()`	Same as `preempt_enable()`
`put_cpu_no_resched()`	Same as `preempt_enable_no_resched()`

The `preempt_enable()` macro decreases the preemption counter, then checks whether the `TIF_NEED_RESCHED` flag is set (see Table 4-15 in Chapter 4). In this case, a process switch request is pending, so the macro invokes the `preempt_schedule()` function, which essentially executes the following code:

```
if (!current_thread_info->preempt_count && !irqs_disabled()) {
    current_thread_info->preempt_count = PREEMPT_ACTIVE;
    schedule();
    current_thread_info->preempt_count = 0;
}
```

The function checks whether local interrupts are enabled and the `preempt_count` field of `current` is zero; if both conditions are true, it invokes `schedule()` to select another process to run. Therefore, kernel preemption may happen either when a kernel control path (usually, an interrupt handler) is terminated, or when an exception handler reenables kernel preemption by means of `preempt_enable()`. As we'll see in the section "Disabling and Enabling Deferrable Functions" later in this chapter, kernel preemption may also happen when deferrable functions are enabled.

We'll conclude this section by noticing that kernel preemption introduces a nonnegligible overhead. For that reason, Linux 2.6 features a kernel configuration option that allows users to enable or disable kernel preemption when compiling the kernel.

When Synchronization Is Necessary

Chapter 1 introduced the concepts of race condition and critical region for processes. The same definitions apply to kernel control paths. In this chapter, a race condition can occur when the outcome of a computation depends on how two or more interleaved kernel control paths are nested. A *critical region* is a section of code that must be completely executed by the kernel control path that enters it before another kernel control path can enter it.

Interleaving kernel control paths complicates the life of kernel developers: they must apply special care in order to identify the critical regions in exception handlers, interrupt handlers, deferrable functions, and kernel threads. Once a critical region has

been identified, it must be suitably protected to ensure that any time at most one kernel control path is inside that region.

Suppose, for instance, that two different interrupt handlers need to access the same data structure that contains several related member variables—for instance, a buffer and an integer indicating its length. All statements affecting the data structure must be put into a single critical region. If the system includes a single CPU, the critical region can be implemented by disabling interrupts while accessing the shared data structure, because nesting of kernel control paths can only occur when interrupts are enabled.

On the other hand, if the same data structure is accessed only by the service routines of system calls, and if the system includes a single CPU, the critical region can be implemented quite simply by disabling kernel preemption while accessing the shared data structure.

As you would expect, things are more complicated in multiprocessor systems. Many CPUs may execute kernel code at the same time, so kernel developers cannot assume that a data structure can be safely accessed just because kernel preemption is disabled and the data structure is never addressed by an interrupt, exception, or softirq handler.

We'll see in the following sections that the kernel offers a wide range of different synchronization techniques. It is up to kernel designers to solve each synchronization problem by selecting the most efficient technique.

When Synchronization Is Not Necessary

Some design choices already discussed in the previous chapter simplify somewhat the synchronization of kernel control paths. Let us recall them briefly:

- All interrupt handlers acknowledge the interrupt on the PIC and also disable the IRQ line. Further occurrences of the same interrupt cannot occur until the handler terminates.
- Interrupt handlers, softirqs, and tasklets are both nonpreemptable and nonblocking, so they cannot be suspended for a long time interval. In the worst case, their execution will be slightly delayed, because other interrupts occur during their execution (nested execution of kernel control paths).
- A kernel control path performing interrupt handling cannot be interrupted by a kernel control path executing a deferrable function or a system call service routine.
- Softirqs and tasklets cannot be interleaved on a given CPU.
- The same tasklet cannot be executed simultaneously on several CPUs.

Each of the above design choices can be viewed as a constraint that can be exploited to code some kernel functions more easily. Here are a few examples of possible simplifications:

- Interrupt handlers and tasklets need not to be coded as reentrant functions.

- Per-CPU variables accessed by softirqs and tasklets only do not require synchronization.
- A data structure accessed by only one kind of tasklet does not require synchronization.

The rest of this chapter describes what to do when synchronization *is* necessary—i.e., how to prevent data corruption due to unsafe accesses to shared data structures.

Synchronization Primitives

We now examine how kernel control paths can be interleaved while avoiding race conditions among shared data. Table 5-2 lists the synchronization techniques used by the Linux kernel. The "Scope" column indicates whether the synchronization technique applies to all CPUs in the system or to a single CPU. For instance, local interrupt disabling applies to just one CPU (other CPUs in the system are not affected); conversely, an atomic operation affects all CPUs in the system (atomic operations on several CPUs cannot interleave while accessing the same data structure).

Table 5-2. Various types of synchronization techniques used by the kernel

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Let's now briefly discuss each synchronization technique. In the later section "Synchronizing Accesses to Kernel Data Structures," we show how these synchronization techniques can be combined to effectively protect kernel data structures.

Per-CPU Variables

The best synchronization technique consists in designing the kernel so as to avoid the need for synchronization in the first place. As we'll see, in fact, every explicit synchronization primitive has a significant performance cost.

The simplest and most efficient synchronization technique consists of declaring kernel variables as *per-CPU variables*. Basically, a per-CPU variable is an array of data structures, one element per each CPU in the system.

A CPU should not access the elements of the array corresponding to the other CPUs; on the other hand, it can freely read and modify its own element without fear of race conditions, because it is the only CPU entitled to do so. This also means, however, that the per-CPU variables can be used only in particular cases—basically, when it makes sense to logically split the data across the CPUs of the system.

The elements of the per-CPU array are aligned in main memory so that each data structure falls on a different line of the hardware cache (see the section "Hardware Cache" in Chapter 2). Therefore, concurrent accesses to the per-CPU array do not result in cache line snooping and invalidation, which are costly operations in terms of system performance.

While per-CPU variables provide protection against concurrent accesses from several CPUs, they do not provide protection against accesses from asynchronous functions (interrupt handlers and deferrable functions). In these cases, additional synchronization primitives are required.

Furthermore, per-CPU variables are prone to race conditions caused by kernel preemption, both in uniprocessor and multiprocessor systems. As a general rule, a kernel control path should access a per-CPU variable with kernel preemption disabled. Just consider, for instance, what would happen if a kernel control path gets the address of its local copy of a per-CPU variable, and then it is preempted and moved to another CPU: the address still refers to the element of the previous CPU.

Table 5-3 lists the main functions and macros offered by the kernel to use per-CPU variables.

Table 5-3. Functions and macros for the per-CPU variables

Macro or function name	Description
`DEFINE_PER_CPU(type, name)`	Statically allocates a per-CPU array called `name` of `type` data structures
`per_cpu(name, cpu)`	Selects the element for CPU `cpu` of the per-CPU array `name`
`__get_cpu_var(name)`	Selects the local CPU's element of the per-CPU array `name`
`get_cpu_var(name)`	Disables kernel preemption, then selects the local CPU's element of the per-CPU array `name`
`put_cpu_var(name)`	Enables kernel preemption (`name` is not used)
`alloc_percpu(type)`	Dynamically allocates a per-CPU array of `type` data structures and returns its address
`free_percpu(pointer)`	Releases a dynamically allocated per-CPU array at address `pointer`
`per_cpu_ptr(pointer, cpu)`	Returns the address of the element for CPU `cpu` of the per-CPU array at address `pointer`

Atomic Operations

Several assembly language instructions are of type "read-modify-write"—that is, they access a memory location twice, the first time to read the old value and the second time to write a new value.

Suppose that two kernel control paths running on two CPUs try to "read-modify-write" the same memory location at the same time by executing nonatomic operations. At first, both CPUs try to read the same location, but the *memory arbiter* (a hardware circuit that serializes accesses to the RAM chips) steps in to grant access to one of them and delay the other. However, when the first read operation has completed, the delayed CPU reads exactly the same (old) value from the memory location. Both CPUs then try to write the same (new) value to the memory location; again, the bus memory access is serialized by the memory arbiter, and eventually both write operations succeed. However, the global result is incorrect because both CPUs write the same (new) value. Thus, the two interleaving "read-modify-write" operations act as a single one.

The easiest way to prevent race conditions due to "read-modify-write" instructions is by ensuring that such operations are atomic at the chip level. Every such operation must be executed in a single instruction without being interrupted in the middle and avoiding accesses to the same memory location by other CPUs. These very small *atomic operations* can be found at the base of other, more flexible mechanisms to create critical regions.

Let's review 80×86 instructions according to that classification:

- Assembly language instructions that make zero or one aligned memory access are atomic.*
- Read-modify-write assembly language instructions (such as `inc` or `dec`) that read data from memory, update it, and write the updated value back to memory are atomic if no other processor has taken the memory bus after the read and before the write. Memory bus stealing never happens in a uniprocessor system.
- Read-modify-write assembly language instructions whose opcode is prefixed by the `lock` byte (`0xf0`) are atomic even on a multiprocessor system. When the control unit detects the prefix, it "locks" the memory bus until the instruction is finished. Therefore, other processors cannot access the memory location while the locked instruction is being executed.
- Assembly language instructions whose opcode is prefixed by a `rep` byte (`0xf2`, `0xf3`, which forces the control unit to repeat the same instruction several times) are not atomic. The control unit checks for pending interrupts before executing a new iteration.

When you write C code, you cannot guarantee that the compiler will use an atomic instruction for an operation like `a=a+1` or even for `a++`. Thus, the Linux kernel provides a special `atomic_t` type (an atomically accessible counter) and some special

* A data item is aligned in memory when its address is a multiple of its size in bytes. For instance, the address of an aligned short integer must be a multiple of two, while the address of an aligned integer must be a multiple of four. Generally speaking, an unaligned memory access is not atomic.

functions and macros (see Table 5-4) that act on `atomic_t` variables and are implemented as single, atomic assembly language instructions. On multiprocessor systems, each such instruction is prefixed by a `lock` byte.

Table 5-4. Atomic operations in Linux

Function	Description
`atomic_read(v)`	Return `*v`
`atomic_set(v,i)`	Set `*v` to `i`
`atomic_add(i,v)`	Add `i` to `*v`
`atomic_sub(i,v)`	Subtract `i` from `*v`
`atomic_sub_and_test(i, v)`	Subtract `i` from `*v` and return 1 if the result is zero; 0 otherwise
`atomic_inc(v)`	Add 1 to `*v`
`atomic_dec(v)`	Subtract 1 from `*v`
`atomic_dec_and_test(v)`	Subtract 1 from `*v` and return 1 if the result is zero; 0 otherwise
`atomic_inc_and_test(v)`	Add 1 to `*v` and return 1 if the result is zero; 0 otherwise
`atomic_add_negative(i, v)`	Add `i` to `*v` and return 1 if the result is negative; 0 otherwise
`atomic_inc_return(v)`	Add 1 to `*v` and return the new value of `*v`
`atomic_dec_return(v)`	Subtract 1 from `*v` and return the new value of `*v`
`atomic_add_return(i, v)`	Add `i` to `*v` and return the new value of `*v`
`atomic_sub_return(i, v)`	Subtract `i` from `*v` and return the new value of `*v`

Another class of atomic functions operate on bit masks (see Table 5-5).

Table 5-5. Atomic bit handling functions in Linux

Function	Description
`test_bit(nr, addr)`	Return the value of the `nr`th bit of `*addr`
`set_bit(nr, addr)`	Set the `nr`th bit of `*addr`
`clear_bit(nr, addr)`	Clear the `nr`th bit of `*addr`
`change_bit(nr, addr)`	Invert the `nr`th bit of `*addr`
`test_and_set_bit(nr, addr)`	Set the `nr`th bit of `*addr` and return its old value
`test_and_clear_bit(nr, addr)`	Clear the `nr`th bit of `*addr` and return its old value
`test_and_change_bit(nr, addr)`	Invert the `nr`th bit of `*addr` and return its old value
`atomic_clear_mask(mask, addr)`	Clear all bits of `*addr` specified by `mask`
`atomic_set_mask(mask, addr)`	Set all bits of `*addr` specified by `mask`

Optimization and Memory Barriers

When using optimizing compilers, you should never take for granted that instructions will be performed in the exact order in which they appear in the source code. For example, a compiler might reorder the assembly language instructions in such a

way to optimize how registers are used. Moreover, modern CPUs usually execute several instructions in parallel and might reorder memory accesses. These kinds of reordering can greatly speed up the program.

When dealing with synchronization, however, reordering instructions must be avoided. Things would quickly become hairy if an instruction placed after a synchronization primitive is executed before the synchronization primitive itself. Therefore, all synchronization primitives act as optimization and memory barriers.

An *optimization barrier* primitive ensures that the assembly language instructions corresponding to C statements placed before the primitive are not mixed by the compiler with assembly language instructions corresponding to C statements placed after the primitive. In Linux the `barrier()` macro, which expands into `asm volatile("":::"memory")`, acts as an optimization barrier. The `asm` instruction tells the compiler to insert an assembly language fragment (empty, in this case). The `volatile` keyword forbids the compiler to reshuffle the `asm` instruction with the other instructions of the program. The `memory` keyword forces the compiler to assume that all memory locations in RAM have been changed by the assembly language instruction; therefore, the compiler cannot optimize the code by using the values of memory locations stored in CPU registers before the `asm` instruction. Notice that the optimization barrier does not ensure that the executions of the assembly language instructions are not mixed by the CPU—this is a job for a memory barrier.

A *memory barrier* primitive ensures that the operations placed before the primitive are finished before starting the operations placed after the primitive. Thus, a memory barrier is like a firewall that cannot be passed by an assembly language instruction.

In the 80×86 processors, the following kinds of assembly language instructions are said to be "serializing" because they act as memory barriers:

- All instructions that operate on I/O ports
- All instructions prefixed by the `lock` byte (see the section "Atomic Operations")
- All instructions that write into control registers, system registers, or debug registers (for instance, `cli` and `sti`, which change the status of the `IF` flag in the `eflags` register)
- The `lfence`, `sfence`, and `mfence` assembly language instructions, which have been introduced in the Pentium 4 microprocessor to efficiently implement read memory barriers, write memory barriers, and read-write memory barriers, respectively.
- A few special assembly language instructions; among them, the `iret` instruction that terminates an interrupt or exception handler

Linux uses a few memory barrier primitives, which are shown in Table 5-6. These primitives act also as optimization barriers, because we must make sure the compiler does not move the assembly language instructions around the barrier. "Read memory

barriers" act only on instructions that read from memory, while "write memory barriers" act only on instructions that write to memory. Memory barriers can be useful in both multiprocessor and uniprocessor systems. The `smp_xxx()` primitives are used whenever the memory barrier should prevent race conditions that might occur only in multiprocessor systems; in uniprocessor systems, they do nothing. The other memory barriers are used to prevent race conditions occurring both in uniprocessor and multiprocessor systems.

Table 5-6. Memory barriers in Linux

Macro	Description
`mb()`	Memory barrier for MP and UP
`rmb()`	Read memory barrier for MP and UP
`wmb()`	Write memory barrier for MP and UP
`smp_mb()`	Memory barrier for MP only
`smp_rmb()`	Read memory barrier for MP only
`smp_wmb()`	Write memory barrier for MP only

The implementations of the memory barrier primitives depend on the architecture of the system. On an 80×86 microprocessor, the `rmb()` macro usually expands into `asm volatile("lfence")` if the CPU supports the `lfence` assembly language instruction, or into `asm volatile("lock;addl $0,0(%%esp)":::"memory")` otherwise. The `asm` statement inserts an assembly language fragment in the code generated by the compiler and acts as an optimization barrier. The `lock; addl $0,0(%%esp)` assembly language instruction adds zero to the memory location on top of the stack; the instruction is useless by itself, but the `lock` prefix makes the instruction a memory barrier for the CPU.

The `wmb()` macro is actually simpler because it expands into `barrier()`. This is because existing Intel microprocessors never reorder write memory accesses, so there is no need to insert a serializing assembly language instruction in the code. The macro, however, forbids the compiler from shuffling the instructions.

Notice that in multiprocessor systems, all atomic operations described in the earlier section "Atomic Operations" act as memory barriers because they use the `lock` byte.

Spin Locks

A widely used synchronization technique is *locking*. When a kernel control path must access a shared data structure or enter a critical region, it needs to acquire a "lock" for it. A resource protected by a locking mechanism is quite similar to a resource confined in a room whose door is locked when someone is inside. If a kernel control path wishes to access the resource, it tries to "open the door" by acquiring the lock. It succeeds only if the resource is free. Then, as long as it wants to use the resource, the door remains locked. When the kernel control path releases the lock, the door is unlocked and another kernel control path may enter the room.

Figure 5-1 illustrates the use of locks. Five kernel control paths (P0, P1, P2, 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 because no kernel control path needs to enter it.

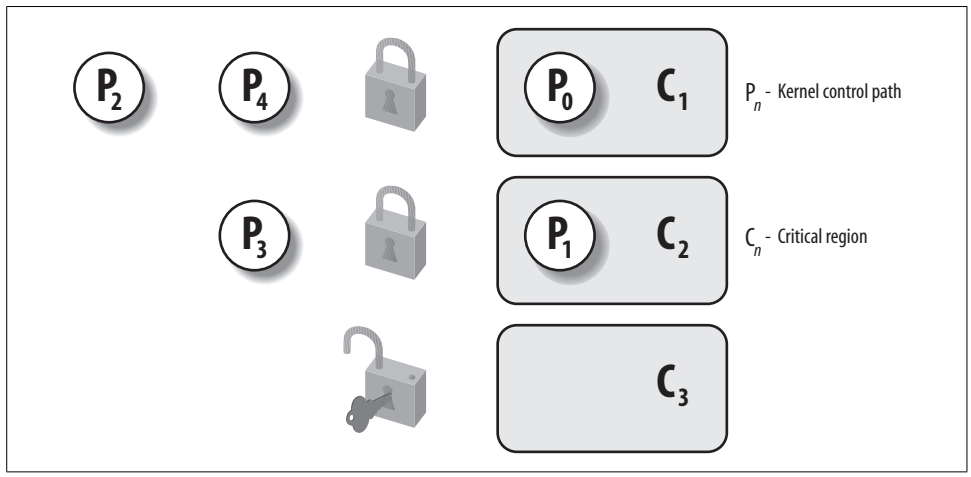


Figure 5-1. Protecting critical regions with several locks

Spin locks are a special kind of lock designed to work in a multiprocessor environment. If the kernel control path finds the spin lock "open," it acquires the lock and continues its execution. Conversely, if the kernel control path finds the lock "closed" by a kernel control path running on another CPU, it "spins" around, repeatedly executing a tight instruction loop, until the lock is released.

The instruction loop of spin locks represents a "busy wait." The waiting kernel control path keeps running on the CPU, even if it has nothing to do besides waste time. Nevertheless, spin locks are usually convenient, because many kernel resources are locked for a fraction of a millisecond only; therefore, it would be far more time-consuming to release the CPU and reacquire it later.

As a general rule, kernel preemption is disabled in every critical region protected by spin locks. In the case of a uniprocessor system, the locks themselves are useless, and the spin lock primitives just disable or enable the kernel preemption. Please notice that kernel preemption is still enabled during the busy wait phase, thus a process waiting for a spin lock to be released could be replaced by a higher priority process.

In Linux, each spin lock is represented by a `spinlock_t` structure consisting of two fields:

`slock`
: Encodes the spin lock state: the value 1 corresponds to the unlocked state, while every negative value and 0 denote the locked state

`break_lock`
: Flag signaling that a process is busy waiting for the lock (present only if the kernel supports both SMP and kernel preemption)

Six macros shown in Table 5-7 are used to initialize, test, and set spin locks. All these macros are based on atomic operations; this ensures that the spin lock will be updated properly even when multiple processes running on different CPUs try to modify the lock at the same time.*

Table 5-7. Spin lock macros

Macro	Description
`spin_lock_init()`	Set the spin lock to 1 (unlocked)
`spin_lock()`	Cycle until spin lock becomes 1 (unlocked), then set it to 0 (locked)
`spin_unlock()`	Set the spin lock to 1 (unlocked)
`spin_unlock_wait()`	Wait until the spin lock becomes 1 (unlocked)
`spin_is_locked()`	Return 0 if the spin lock is set to 1 (unlocked); 1 otherwise
`spin_trylock()`	Set the spin lock to 0 (locked), and return 1 if the previous value of the lock was 1; 0 otherwise

The spin_lock macro with kernel preemption

Let's discuss in detail the `spin_lock` macro, which is used to acquire a spin lock. The following description refers to a preemptive kernel for an SMP system. The macro takes the address `slp` of the spin lock as its parameter and executes the following actions:

1. Invokes `preempt_disable()` to disable kernel preemption.
2. Invokes the `_raw_spin_trylock()` function, which does an atomic test-and-set operation on the spin lock's `slock` field; this function executes first some instructions equivalent to the following assembly language fragment:

   ```
   movb $0, %al
   xchgb %al, slp->slock
   ```

   The `xchg` assembly language instruction exchanges atomically the content of the 8-bit `%al` register (storing zero) with the content of the memory location pointed to by `slp->slock`. The function then returns the value 1 if the old value stored in the spin lock (in `%al` after the `xchg` instruction) was positive, the value 0 otherwise.
3. If the old value of the spin lock was positive, the macro terminates: the kernel control path has acquired the spin lock.

* Spin locks, ironically enough, are global and therefore must themselves be protected against concurrent accesses.

4. Otherwise, the kernel control path failed in acquiring the spin lock, thus the macro must cycle until the spin lock is released by a kernel control path running on some other CPU. Invokes `preempt_enable()` to undo the increase of the preemption counter done in step 1. If kernel preemption was enabled before executing the `spin_lock` macro, another process can now replace this process while it waits for the spin lock.
5. If the `break_lock` field is equal to zero, sets it to one. By checking this field, the process owning the lock and running on another CPU can learn whether there are other processes waiting for the lock. If a process holds a spin lock for a long time, it may decide to release it prematurely to allow another process waiting for the same spin lock to progress.
6. Executes the wait cycle:

   ```
   while (spin_is_locked(slp) && slp->break_lock)
       cpu_relax();
   ```

   The `cpu_relax()` macro reduces to a `pause` assembly language instruction. This instruction has been introduced in the Pentium 4 model to optimize the execution of spin lock loops. By introducing a short delay, it speeds up the execution of code following the lock and reduces power consumption. The `pause` instruction is backward compatible with earlier models of 80×86 microprocessors because it corresponds to the instruction `rep;nop`, that is, to a no-operation.
7. Jumps back to step 1 to try once more to get the spin lock.

The spin_lock macro without kernel preemption

If the kernel preemption option has not been selected when the kernel was compiled, the `spin_lock` macro is quite different from the one described above. In this case, the macro yields a assembly language fragment that is essentially equivalent to the following tight busy wait:*

```
1: lock; decb slp->slock
   jns  3f
2: pause
   cmpb $0,slp->slock
   jle 2b
   jmp 1b
3:
```

The `decb` assembly language instruction decreases the spin lock value; the instruction is atomic because it is prefixed by the `lock` byte. A test is then performed on the sign flag. If it is clear, it means that the spin lock was set to 1 (unlocked), so normal

* The actual implementation of the tight busy wait loop is slightly more complicated. The code at label 2, which is executed only if the spin lock is busy, is included in an auxiliary section so that in the most frequent case (when the spin lock is already free) the hardware cache is not filled with code that won't be executed. In our discussion, we omit these optimization details.

execution continues at label 3 (the f suffix denotes the fact that the label is a "forward" one; it appears in a later line of the program). Otherwise, the tight loop at label 2 (the b suffix denotes a "backward" label) is executed until the spin lock assumes a positive value. Then execution restarts from label 1, since it is unsafe to proceed without checking whether another processor has grabbed the lock.

The spin_unlock macro

The spin_unlock macro releases a previously acquired spin lock; it essentially executes the assembly language instruction:

```
movb $1, slp->slock
```

and then invokes preempt_enable() (if kernel preemption is not supported, preempt_enable() does nothing). Notice that the lock byte is not used because write-only accesses in memory are always atomically executed by the current 80×86 microprocessors.

Read/Write Spin Locks

Read/write spin locks have been introduced to increase the amount of concurrency inside the kernel. They allow several kernel control paths to simultaneously read the same data structure, as long as no kernel control path modifies it. If a kernel control path wishes to write to the structure, it must acquire the write version of the read/write lock, which grants exclusive access to the resource. Of course, allowing concurrent reads on data structures improves system performance.

Figure 5-2 illustrates two critical regions (C1 and C2) protected by read/write locks. Kernel control paths R0 and R1 are reading the data structures in C1 at the same time, while W0 is waiting to acquire the lock for writing. Kernel control path W1 is writing the data structures in C2, while both R2 and W2 are waiting to acquire the lock for reading and writing, respectively.

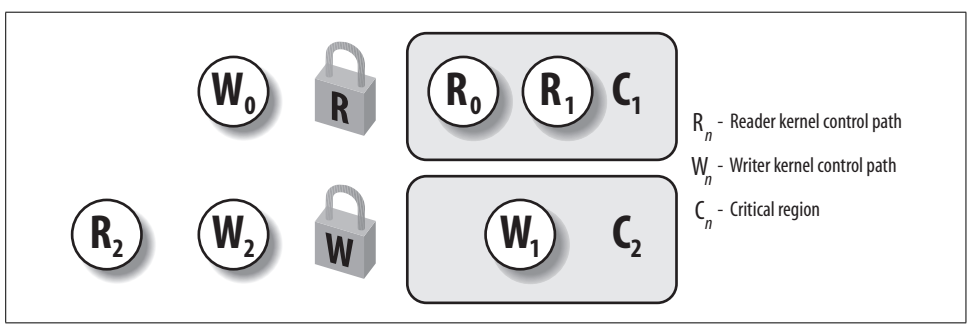


Figure 5-2. Read/write spin locks

Each read/write spin lock is a rwlock_t structure; its lock field is a 32-bit field that encodes two distinct pieces of information:

- A 24-bit counter denoting the number of kernel control paths currently reading the protected data structure. The two's complement value of this counter is stored in bits 0–23 of the field.
- An unlock flag that is set when no kernel control path is reading or writing, and clear otherwise. This unlock flag is stored in bit 24 of the field.

Notice that the lock field stores the number 0x01000000 if the spin lock is idle (unlock flag set and no readers), the number 0x00000000 if it has been acquired for writing (unlock flag clear and no readers), and any number in the sequence 0x00ffffff, 0x00fffffe, and so on, if it has been acquired for reading by one, two, or more processes (unlock flag clear and the two's complement on 24 bits of the number of readers). As the spinlock_t structure, the rwlock_t structure also includes a break_lock field.

The rwlock_init macro initializes the lock field of a read/write spin lock to 0x01000000 (unlocked) and the break_lock field to zero.

Getting and releasing a lock for reading

The read_lock macro, applied to the address rwlp of a read/write spin lock, is similar to the spin_lock macro described in the previous section. If the kernel preemption option has been selected when the kernel was compiled, the macro performs the very same actions as those of spin_lock(), with just one exception: to effectively acquire the read/write spin lock in step 2, the macro executes the _raw_read_trylock() function:

```
int _raw_read_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    atomic_dec(count);
    if (atomic_read(count) >= 0)
        return 1;
    atomic_inc(count);
    return 0;
}
```

The lock field—the read/write lock counter—is accessed by means of atomic operations. Notice, however, that the whole function does not act atomically on the counter: for instance, the counter might change after having tested its value with the if statement and before returning 1. Nevertheless, the function works properly: in fact, the function returns 1 only if the counter was not zero or negative before the decrement, because the counter is equal to 0x01000000 for no owner, 0x00ffffff for one reader, and 0x00000000 for one writer.

If the kernel preemption option has not been selected when the kernel was compiled, the read_lock macro yields the following assembly language code:

```
        movl $rwlp->lock,%eax
        lock; subl $1,(%eax)
        jns 1f
        call __read_lock_failed
    1:
```

where `__read_lock_failed()` is the following assembly language function:

```
    __read_lock_failed:
        lock; incl (%eax)
    1:  pause
        cmpl $1,(%eax)
        js 1b
        lock; decl (%eax)
        js __read_lock_failed
        ret
```

The `read_lock` macro atomically decreases the spin lock value by 1, thus increasing the number of readers. The spin lock is acquired if the decrement operation yields a nonnegative value; otherwise, the `__read_lock_failed()` function is invoked. The function atomically increases the `lock` field to undo the decrement operation performed by the `read_lock` macro, and then loops until the field becomes positive (greater than or equal to 1). Next, `__read_lock_failed()` tries to get the spin lock again (another kernel control path could acquire the spin lock for writing right after the `cmpl` instruction).

Releasing the read lock is quite simple, because the `read_unlock` macro must simply increase the counter in the `lock` field with the assembly language instruction:

```
lock; incl rwlp->lock
```

to decrease the number of readers, and then invoke `preempt_enable()` to reenable kernel preemption.

Getting and releasing a lock for writing

The `write_lock` macro is implemented in the same way as `spin_lock()` and `read_lock()`. For instance, if kernel preemption is supported, the function disables kernel preemption and tries to grab the lock right away by invoking `_raw_write_trylock()`. If this function returns 0, the lock was already taken, thus the macro reenables kernel preemption and starts a busy wait loop, as explained in the description of `spin_lock()` in the previous section.

The `_raw_write_trylock()` function is shown below:

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

The _raw_write_trylock() function subtracts 0x01000000 from the read/write spin lock value, thus clearing the unlock flag (bit 24). If the subtraction operation yields zero (no readers), the lock is acquired and the function returns 1; otherwise, the function atomically adds 0x01000000 to the spin lock value to undo the subtraction operation.

Once again, releasing the write lock is much simpler because the write_unlock macro must simply set the unlock flag in the lock field with the assembly language instruction:

```
lock; addl $0x01000000,rwlp
```

and then invoke preempt_enable().

Seqlocks

When using read/write spin locks, requests issued by kernel control paths to perform a read_lock or a write_lock operation have the same priority: readers must wait until the writer has finished and, similarly, a writer must wait until all readers have finished.

Seqlocks introduced in Linux 2.6 are similar to read/write spin locks, except that they give a much higher priority to writers: in fact a writer is allowed to proceed even when readers are active. The good part of this strategy is that a writer never waits (unless another writer is active); the bad part is that a reader may sometimes be forced to read the same data several times until it gets a valid copy.

Each seqlock is a seqlock_t structure consisting of two fields: a lock field of type spinlock_t and an integer sequence field. This second field plays the role of a sequence counter. Each reader must read this sequence counter twice, before and after reading the data, and check whether the two values coincide. In the opposite case, a new writer has become active and has increased the sequence counter, thus implicitly telling the reader that the data just read is not valid.

A seqlock_t variable is initialized to "unlocked" either by assigning to it the value SEQLOCK_UNLOCKED, or by executing the seqlock_init macro. Writers acquire and release a seqlock by invoking write_seqlock() and write_sequnlock(). The first function acquires the spin lock in the seqlock_t data structure, then increases the sequence counter by one; the second function increases the sequence counter once more, then releases the spin lock. This ensures that when the writer is in the middle of writing, the counter is odd, and that when no writer is altering data, the counter is even. Readers implement a critical region as follows:

```
unsigned int seq;
do {
    seq = read_seqbegin(&seqlock);
    /* ... CRITICAL REGION ... */
} while (read_seqretry(&seqlock, seq));
```

`read_seqbegin()` returns the current sequence number of the seqlock; `read_seqretry()` returns 1 if either the value of the `seq` local variable is odd (a writer was updating the data structure when the `read_seqbegin()` function has been invoked), or if the value of `seq` does not match the current value of the seqlock's sequence counter (a writer started working while the reader was still executing the code in the critical region).

Notice that when a reader enters a critical region, it does not need to disable kernel preemption; on the other hand, the writer automatically disables kernel preemption when entering the critical region, because it acquires the spin lock.

Not every kind of data structure can be protected by a seqlock. As a general rule, the following conditions must hold:

- The data structure to be protected does not include pointers that are modified by the writers and dereferenced by the readers (otherwise, a writer could change the pointer under the nose of the readers)
- The code in the critical regions of the readers does not have side effects (otherwise, multiple reads would have different effects from a single read)

Furthermore, the critical regions of the readers should be short and writers should seldom acquire the seqlock, otherwise repeated read accesses would cause a severe overhead. A typical usage of seqlocks in Linux 2.6 consists of protecting some data structures related to the system time handling (see Chapter 6).

Read-Copy Update (RCU)

Read-copy update (*RCU*) is yet another synchronization technique designed to protect data structures that are mostly accessed for reading by several CPUs. RCU allows many readers and many writers to proceed concurrently (an improvement over seqlocks, which allow only one writer to proceed). Moreover, RCU is lock-free, that is, it uses no lock or counter shared by all CPUs; this is a great advantage over read/write spin locks and seqlocks, which have a high overhead due to cache line-snooping and invalidation.

How does RCU obtain the surprising result of synchronizing several CPUs without shared data structures? The key idea consists of limiting the scope of RCU as follows:

1. Only data structures that are dynamically allocated and referenced by means of pointers can be protected by RCU.
2. No kernel control path can sleep inside a critical region protected by RCU.

When a kernel control path wants to read an RCU-protected data structure, it executes the `rcu_read_lock()` macro, which is equivalent to `preempt_disable()`. Next, the reader dereferences the pointer to the data structure and starts reading it. As stated above, the reader cannot sleep until it finishes reading the data structure; the end of the critical region is marked by the `rcu_read_unlock()` macro, which is equivalent to `preempt_enable()`.

Because the reader does very little to prevent race conditions, we could expect that the writer has to work a bit more. In fact, when a writer wants to update the data structure, it dereferences the pointer and makes a copy of the whole data structure. Next, the writer modifies the copy. Once finished, the writer changes the pointer to the data structure so as to make it point to the updated copy. Because changing the value of the pointer is an atomic operation, each reader or writer sees either the old copy or the new one: no corruption in the data structure may occur. However, a memory barrier is required to ensure that the updated pointer is seen by the other CPUs only after the data structure has been modified. Such a memory barrier is implicitly introduced if a spin lock is coupled with RCU to forbid the concurrent execution of writers.

The real problem with the RCU technique, however, is that the old copy of the data structure cannot be freed right away when the writer updates the pointer. In fact, the readers that were accessing the data structure when the writer started its update could still be reading the old copy. The old copy can be freed only after all (potential) readers on the CPUs have executed the `rcu_read_unlock()` macro. The kernel requires every potential reader to execute that macro before:

- The CPU performs a process switch (see restriction 2 earlier).
- The CPU starts executing in User Mode.
- The CPU executes the idle loop (see the section "Kernel Threads" in Chapter 3).

In each of these cases, we say that the CPU has gone through a *quiescent state*.

The `call_rcu()` function is invoked by the writer to get rid of the old copy of the data structure. It receives as its parameters the address of an `rcu_head` descriptor (usually embedded inside the data structure to be freed) and the address of a *callback* function to be invoked when all CPUs have gone through a quiescent state. Once executed, the callback function usually frees the old copy of the data structure.

The `call_rcu()` function stores in the `rcu_head` descriptor the address of the callback and its parameter, then inserts the descriptor in a per-CPU list of callbacks. Periodically, once every tick (see the section "Updating Local CPU Statistics" in Chapter 6), the kernel checks whether the local CPU has gone through a quiescent state. When all CPUs have gone through a quiescent state, a local tasklet—whose descriptor is stored in the `rcu_tasklet` per-CPU variable—executes all callbacks in the list.

RCU is a new addition in Linux 2.6; it is used in the networking layer and in the Virtual Filesystem.

Semaphores

We have already introduced semaphores in the section "Synchronization and Critical Regions" in Chapter 1. Essentially, they implement a locking primitive that allows waiters to sleep until the desired resource becomes free.

Actually, Linux offers two kinds of semaphores:

- Kernel semaphores, which are used by kernel control paths
- System V IPC semaphores, which are used by User Mode processes

In this section, we focus on kernel semaphores, while IPC semaphores are described in Chapter 19.

A kernel semaphore is similar to a spin lock, in that it doesn't allow a kernel control path to proceed unless the lock is open. However, whenever a kernel control path tries to acquire a busy resource protected by a kernel semaphore, the corresponding process is suspended. It becomes runnable again when the resource is released. Therefore, kernel semaphores can be acquired only by functions that are allowed to sleep; interrupt handlers and deferrable functions cannot use them.

A kernel semaphore is an object of type `struct semaphore`, containing the fields shown in the following list.

`count`
: Stores an `atomic_t` value. If it is greater than 0, the resource is free—that is, it is currently available. If `count` is equal to 0, the semaphore is busy but no other process is waiting for the protected resource. Finally, if `count` is negative, the resource is unavailable and at least one process is waiting for it.

`wait`
: Stores the address of a wait queue list that includes all sleeping processes that are currently waiting for the resource. Of course, if `count` is greater than or equal to 0, the wait queue is empty.

`sleepers`
: Stores a flag that indicates whether some processes are sleeping on the semaphore. We'll see this field in operation soon.

The `init_MUTEX()` and `init_MUTEX_LOCKED()` functions may be used to initialize a semaphore for exclusive access: they set the `count` field to 1 (free resource with exclusive access) and 0 (busy resource with exclusive access currently granted to the process that initializes the semaphore), respectively. The `DECLARE_MUTEX` and `DECLARE_MUTEX_LOCKED` macros do the same, but they also statically allocate the `struct semaphore` variable. Note that a semaphore could also be initialized with an arbitrary positive value *n* for `count`. In this case, at most *n* processes are allowed to concurrently access the resource.

Getting and releasing semaphores

Let's start by discussing how to release a semaphore, which is much simpler than getting one. When a process wishes to release a kernel semaphore lock, it invokes the

up() function. This function is essentially equivalent to the following assembly language fragment:

```
        movl $sem->count,%ecx
        lock; incl (%ecx)
        jg 1f
        lea %ecx,%eax
        pushl %edx
        pushl %ecx
        call __up
        popl %ecx
        popl %edx
    1:
```

where __up() is the following C function:

```
__attribute__((regparm(3))) void __up(struct semaphore *sem)
{
    wake_up(&sem->wait);
}
```

The up() function increases the count field of the *sem semaphore, and then it checks whether its value is greater than 0. The increment of count and the setting of the flag tested by the following jump instruction must be atomically executed, or else another kernel control path could concurrently access the field value, with disastrous results. If count is greater than 0, there was no process sleeping in the wait queue, so nothing has to be done. Otherwise, the __up() function is invoked so that one sleeping process is woken up. Notice that __up() receives its parameter from the eax register (see the description of the __switch_to() function in the section "Performing the Process Switch" in Chapter 3).

Conversely, when a process wishes to acquire a kernel semaphore lock, it invokes the down() function. The implementation of down() is quite involved, but it is essentially equivalent to the following:

```
down:
    movl $sem->count,%ecx
    lock; decl (%ecx);
    jns 1f
    lea %ecx, %eax
    pushl %edx
    pushl %ecx
    call __down
    popl %ecx
    popl %edx
1:
```

where __down() is the following C function:

```
__attribute__((regparm(3))) void __down(struct semaphore * sem)
{
    DECLARE_WAITQUEUE(wait, current);
    unsigned long flags;
    current->state = TASK_UNINTERRUPTIBLE;
```

```
        spin_lock_irqsave(&sem->wait.lock, flags);
        add_wait_queue_exclusive_locked(&sem->wait, &wait);
        sem->sleepers++;
        for (;;) {
            if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
                sem->sleepers = 0;
                break;
            }
            sem->sleepers = 1;
            spin_unlock_irqrestore(&sem->wait.lock, flags);
            schedule();
            spin_lock_irqsave(&sem->wait.lock, flags);
            current->state = TASK_UNINTERRUPTIBLE;
        }
        remove_wait_queue_locked(&sem->wait, &wait);
        wake_up_locked(&sem->wait);
        spin_unlock_irqrestore(&sem->wait.lock, flags);
        current->state = TASK_RUNNING;
    }
```

The `down()` function decreases the `count` field of the `*sem` semaphore, and then checks whether its value is negative. Again, the decrement and the test must be atomically executed. If `count` is greater than or equal to 0, the current process acquires the resource and the execution continues normally. Otherwise, `count` is negative, and the current process must be suspended. The contents of some registers are saved on the stack, and then `__down()` is invoked.

Essentially, the `__down()` function changes the state of the current process from `TASK_RUNNING` to `TASK_UNINTERRUPTIBLE`, and it puts the process in the semaphore wait queue. Before accessing the fields of the `semaphore` structure, the function also gets the `sem->wait.lock` spin lock that protects the semaphore wait queue (see "How Processes Are Organized" in Chapter 3) and disables local interrupts. Usually, wait queue functions get and release the wait queue spin lock as necessary when inserting and deleting an element. The `__down()` function, however, uses the wait queue spin lock also to protect the other fields of the `semaphore` data structure, so that no process running on another CPU is able to read or modify them. To that end, `__down()` uses the "`_locked`" versions of the wait queue functions, which assume that the spin lock has been already acquired before their invocations.

The main task of the `__down()` function is to suspend the current process until the semaphore is released. However, the way in which this is done is quite involved. To easily understand the code, keep in mind that the `sleepers` field of the semaphore is usually set to 0 if no process is sleeping in the wait queue of the semaphore, and it is set to 1 otherwise. Let's try to explain the code by considering a few typical cases.

MUTEX semaphore open (`count` *equal to 1,* `sleepers` *equal to 0)*

The `down` macro just sets the `count` field to 0 and jumps to the next instruction of the main program; therefore, the `__down()` function is not executed at all.

MUTEX semaphore closed, no sleeping processes (`count` *equal to 0,* `sleepers` *equal to 0)*

The `down` macro decreases `count` and invokes the `__down()` function with the `count` field set to –1 and the `sleepers` field set to 0. In each iteration of the loop, the function checks whether the `count` field is negative. (Observe that the `count` field is not changed by `atomic_add_negative()` because `sleepers` is equal to 0 when the function is invoked.)

- If the `count` field is negative, the function invokes `schedule()` to suspend the current process. The `count` field is still set to –1, and the `sleepers` field to 1. The process picks up its run subsequently inside this loop and issues the test again.
- If the `count` field is not negative, the function sets `sleepers` to 0 and exits from the loop. It tries to wake up another process in the semaphore wait queue (but in our scenario, the queue is now empty) and terminates holding the semaphore. On exit, both the `count` field and the `sleepers` field are set to 0, as required when the semaphore is closed but no process is waiting for it.

MUTEX semaphore closed, other sleeping processes (`count` *equal to –1,* `sleepers` *equal to 1)*

The `down` macro decreases `count` and invokes the `__down()` function with `count` set to –2 and `sleepers` set to 1. The function temporarily sets `sleepers` to 2, and then undoes the decrement performed by the `down` macro by adding the value `sleepers`–1 to `count`. At the same time, the function checks whether `count` is still negative (the semaphore could have been released by the holding process right before `__down()` entered the critical region).

- If the `count` field is negative, the function resets `sleepers` to 1 and invokes `schedule()` to suspend the current process. The `count` field is still set to –1, and the `sleepers` field to 1.
- If the `count` field is not negative, the function sets `sleepers` to 0, tries to wake up another process in the semaphore wait queue, and exits holding the semaphore. On exit, the `count` field is set to 0 and the `sleepers` field to 0. The values of both fields look wrong, because there are other sleeping processes. However, consider that another process in the wait queue has been woken up. This process does another iteration of the loop; the `atomic_add_negative()` function subtracts 1 from `count`, restoring it to –1; moreover, before returning to sleep, the woken-up process resets `sleepers` to 1.

So, the code properly works in all cases. Consider that the `wake_up()` function in `__down()` wakes up at most one process, because the sleeping processes in the wait queue are exclusive (see the section "How Processes Are Organized" in Chapter 3).

Only exception handlers, and particularly system call service routines, can use the `down()` function. Interrupt handlers or deferrable functions must not invoke `down()`, because this function suspends the process when the semaphore is busy. For this reason, Linux provides the `down_trylock()` function, which may be safely used by one

of the previously mentioned asynchronous functions. It is identical to down( ) except when the resource is busy. In this case, the function returns immediately instead of putting the process to sleep.

A slightly different function called down_interruptible( ) is also defined. It is widely used by device drivers, because it allows processes that receive a signal while being blocked on a semaphore to give up the "down" operation. If the sleeping process is woken up by a signal before getting the needed resource, the function increases the count field of the semaphore and returns the value -EINTR. On the other hand, if down_interruptible( ) runs to normal completion and gets the resource, it returns 0. The device driver may thus abort the I/O operation when the return value is -EINTR.

Finally, because processes usually find semaphores in an open state, the semaphore functions are optimized for this case. In particular, the up( ) function does not execute jump instructions if the semaphore wait queue is empty; similarly, the down( ) function does not execute jump instructions if the semaphore is open. Much of the complexity of the semaphore implementation is precisely due to the effort of avoiding costly instructions in the main branch of the execution flow.

Read/Write Semaphores

Read/write semaphores are similar to the read/write spin locks described earlier in the section "Read/Write Spin Locks," except that waiting processes are suspended instead of spinning until the semaphore becomes open again.

Many kernel control paths may concurrently acquire a read/write semaphore for reading; however, every writer kernel control path must have exclusive access to the protected resource. Therefore, the semaphore can be acquired for writing only if no other kernel control path is holding it for either read or write access. Read/write semaphores improve the amount of concurrency inside the kernel and improve overall system performance.

The kernel handles all processes waiting for a read/write semaphore in strict FIFO order. Each reader or writer that finds the semaphore closed is inserted in the last position of a semaphore's wait queue list. When the semaphore is released, the process in the first position of the wait queue list are checked. The first process is always awoken. If it is a writer, the other processes in the wait queue continue to sleep. If it is a reader, all readers at the start of the queue, up to the first writer, are also woken up and get the lock. However, readers that have been queued after a writer continue to sleep.

Each read/write semaphore is described by a rw_semaphore structure that includes the following fields:

count
: Stores two 16-bit counters. The counter in the most significant word encodes in two's complement form the sum of the number of nonwaiting writers (either 0

or 1) and the number of waiting kernel control paths. The counter in the less significant word encodes the total number of nonwaiting readers and writers.

`wait_list`
: Points to a list of waiting processes. Each element in this list is a `rwsem_waiter` structure, including a pointer to the descriptor of the sleeping process and a flag indicating whether the process wants the semaphore for reading or for writing.

`wait_lock`
: A spin lock used to protect the wait queue list and the `rw_semaphore` structure itself.

The `init_rwsem()` function initializes an `rw_semaphore` structure by setting the `count` field to 0, the `wait_lock` spin lock to unlocked, and `wait_list` to the empty list.

The `down_read()` and `down_write()` functions acquire the read/write semaphore for reading and writing, respectively. Similarly, the `up_read()` and `up_write()` functions release a read/write semaphore previously acquired for reading and for writing. The `down_read_trylock()` and `down_write_trylock()` functions are similar to `down_read()` and `down_write()`, respectively, but they do not block the process if the semaphore is busy. Finally, the `downgrade_write()` function atomically transforms a write lock into a read lock. The implementation of these five functions is long, but easy to follow because it resembles the implementation of normal semaphores; therefore, we avoid describing them.

Completions

Linux 2.6 also makes use of another synchronization primitive similar to semaphores: *completions*. They have been introduced to solve a subtle race condition that occurs in multiprocessor systems when process A allocates a temporary semaphore variable, initializes it as closed MUTEX, passes its address to process B, and then invokes `down()` on it. Process A plans to destroy the semaphore as soon as it awakens. Later on, process B running on a different CPU invokes `up()` on the semaphore. However, in the current implementation `up()` and `down()` can execute concurrently on the same semaphore. Thus, process A can be woken up and destroy the temporary semaphore while process B is still executing the `up()` function. As a result, `up()` might attempt to access a data structure that no longer exists.

Of course, it is possible to change the implementation of `down()` and `up()` to forbid concurrent executions on the same semaphore. However, this change would require additional instructions, which is a bad thing to do for functions that are so heavily used.

The completion is a synchronization primitive that is specifically designed to solve this problem. The `completion` data structure includes a wait queue head and a flag:

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

The function corresponding to up() is called complete(). It receives as an argument the address of a completion data structure, invokes spin_lock_irqsave() on the spin lock of the completion's wait queue, increases the done field, wakes up the exclusive process sleeping in the wait wait queue, and finally invokes spin_unlock_irqrestore().

The function corresponding to down() is called wait_for_completion(). It receives as an argument the address of a completion data structure and checks the value of the done flag. If it is greater than zero, wait_for_completion() terminates, because complete() has already been executed on another CPU. Otherwise, the function adds current to the tail of the wait queue as an exclusive process and puts current to sleep in the TASK_UNINTERRUPTIBLE state. Once woken up, the function removes current from the wait queue. Then, the function checks the value of the done flag: if it is equal to zero the function terminates, otherwise, the current process is suspended again. As in the case of the complete() function, wait_for_completion() makes use of the spin lock in the completion's wait queue.

The real difference between completions and semaphores is how the spin lock included in the wait queue is used. In completions, the spin lock is used to ensure that complete() and wait_for_completion() cannot execute concurrently. In semaphores, the spin lock is used to avoid letting concurrent down()'s functions mess up the semaphore data structure.

Local Interrupt Disabling

Interrupt disabling is one of the key mechanisms used to ensure that a sequence of kernel statements is treated as a critical section. It allows a kernel control path to continue executing even when hardware devices issue IRQ signals, thus providing an effective way to protect data structures that are also accessed by interrupt handlers. By itself, however, local interrupt disabling does not protect against concurrent accesses to data structures by interrupt handlers running on other CPUs, so in multiprocessor systems, local interrupt disabling is often coupled with spin locks (see the later section "Synchronizing Accesses to Kernel Data Structures").

The local_irq_disable() macro, which makes use of the cli assembly language instruction, disables interrupts on the local CPU. The local_irq_enable() macro, which makes use of the of the sti assembly language instruction, enables them. As stated in the section "IRQs and Interrupts" in Chapter 4, the cli and sti assembly language instructions, respectively, clear and set the IF flag of the eflags control register. The irqs_disabled() macro yields the value one if the IF flag of the eflags register is clear, the value one if the flag is set.

When the kernel enters a critical section, it disables interrupts by clearing the IF flag of the eflags register. But at the end of the critical section, often the kernel can't simply set the flag again. Interrupts can execute in nested fashion, so the kernel does not necessarily know what the IF flag was before the current control path executed. In these cases, the control path must save the old setting of the flag and restore that setting at the end.

Saving and restoring the eflags content is achieved by means of the local_irq_save and local_irq_restore macros, respectively. The local_irq_save macro copies the content of the eflags register into a local variable; the IF flag is then cleared by a cli assembly language instruction. At the end of the critical region, the macro local_irq_restore restores the original content of eflags; therefore, interrupts are enabled only if they were enabled before this control path issued the cli assembly language instruction.

Disabling and Enabling Deferrable Functions

In the section "Softirqs" in Chapter 4, we explained that deferrable functions can be executed at unpredictable times (essentially, on termination of hardware interrupt handlers). Therefore, data structures accessed by deferrable functions must be protected against race conditions.

A trivial way to forbid deferrable functions execution on a CPU is to disable interrupts on that CPU. Because no interrupt handler can be activated, softirq actions cannot be started asynchronously.

As we'll see in the next section, however, the kernel sometimes needs to disable deferrable functions without disabling interrupts. Local deferrable functions can be enabled or disabled on the local CPU by acting on the softirq counter stored in the preempt_count field of the current's thread_info descriptor.

Recall that the do_softirq() function never executes the softirqs if the softirq counter is positive. Moreover, tasklets are implemented on top of softirqs, so setting this counter to a positive value disables the execution of all deferrable functions on a given CPU, not just softirqs.

The local_bh_disable macro adds one to the softirq counter of the local CPU, while the local_bh_enable() function subtracts one from it. The kernel can thus use several nested invocations of local_bh_disable; deferrable functions will be enabled again only by the local_bh_enable macro matching the first local_bh_disable invocation.

After having decreased the softirq counter, local_bh_enable() performs two important operations that help to ensure timely execution of long-waiting threads:

1. Checks the hardirq counter and the softirq counter in the preempt_count field of the local CPU; if both of them are zero and there are pending softirqs to be executed, invokes do_softirq() to activate them (see the section "Softirqs" in Chapter 4).
2. Checks whether the TIF_NEED_RESCHED flag of the local CPU is set; if so, a process switch request is pending, thus invokes the preempt_schedule() function (see the section "Kernel Preemption" earlier in this chapter).

Synchronizing Accesses to Kernel Data Structures

A shared data structure can be protected against race conditions by using some of the synchronization primitives shown in the previous section. Of course, system performance may vary considerably, depending on the kind of synchronization primitive selected. Usually, the following rule of thumb is adopted by kernel developers: *always keep the concurrency level as high as possible in the system*.

In turn, the concurrency level in the system depends on two main factors:

- The number of I/O devices that operate concurrently
- The number of CPUs that do productive work

To maximize I/O throughput, interrupts should be disabled for very short periods of time. As described in the section "IRQs and Interrupts" in Chapter 4, when interrupts are disabled, IRQs issued by I/O devices are temporarily ignored by the PIC, and no new activity can start on such devices.

To use CPUs efficiently, synchronization primitives based on spin locks should be avoided whenever possible. When a CPU is executing a tight instruction loop waiting for the spin lock to open, it is wasting precious machine cycles. Even worse, as we have already said, spin locks have negative effects on the overall performance of the system because of their impact on the hardware caches.

Let's illustrate a couple of cases in which synchronization can be achieved while still maintaining a high concurrency level:

- A shared data structure consisting of a single integer value can be updated by declaring it as an `atomic_t` type and by using atomic operations. An atomic operation is faster than spin locks and interrupt disabling, and it slows down only kernel control paths that concurrently access the data structure.
- Inserting an element into a shared linked list is never atomic, because it consists of at least two pointer assignments. Nevertheless, the kernel can sometimes perform this insertion operation without using locks or disabling interrupts. As an example of why this works, we'll consider the case where a system call service routine (see "System Call Handler and Service Routines" in Chapter 10) inserts new elements in a singly linked list, while an interrupt handler or deferrable function asynchronously looks up the list.

  In the C language, insertion is implemented by means of the following pointer assignments:

  ```
  new->next = list_element->next;
  list_element->next = new;
  ```

  In assembly language, insertion reduces to two consecutive atomic instructions. The first instruction sets up the `next` pointer of the `new` element, but it does not modify the list. Thus, if the interrupt handler sees the list between the execution of the first and second instructions, it sees the list without the new element. If

the handler sees the list after the execution of the second instruction, it sees the list with the new element. The important point is that in either case, the list is consistent and in an uncorrupted state. However, this integrity is assured only if the interrupt handler does not modify the list. If it does, the next pointer that was just set within the new element might become invalid.

However, developers must ensure that the order of the two assignment operations cannot be subverted by the compiler or the CPU's control unit; otherwise, if the system call service routine is interrupted by the interrupt handler between the two assignments, the handler finds a corrupted list. Therefore, a write memory barrier primitive is required:

```
new->next = list_element->next;
wmb();
list_element->next = new;
```

Choosing Among Spin Locks, Semaphores, and Interrupt Disabling

Unfortunately, access patterns to most kernel data structures are a lot more complex than the simple examples just shown, and kernel developers are forced to use semaphores, spin locks, interrupts, and softirq disabling. Generally speaking, choosing the synchronization primitives depends on what kinds of kernel control paths access the data structure, as shown in Table 5-8. Remember that whenever a kernel control path acquires a spin lock (as well as a read/write lock, a seqlock, or a RCU "read lock"), disables the local interrupts, or disables the local softirqs, kernel preemption is automatically disabled.

Table 5-8. Protection required by data structures accessed by kernel control paths

Kernel control paths accessing the data structure	UP protection	MP further protection
Exceptions	Semaphore	None
Interrupts	Local interrupt disabling	Spin lock
Deferrable functions	None	None or spin lock (see Table 5-9)
Exceptions + Interrupts	Local interrupt disabling	Spin lock
Exceptions + Deferrable functions	Local softirq disabling	Spin lock
Interrupts + Deferrable functions	Local interrupt disabling	Spin lock
Exceptions + Interrupts + Deferrable functions	Local interrupt disabling	Spin lock

Protecting a data structure accessed by exceptions

When a data structure is accessed only by exception handlers, race conditions are usually easy to understand and prevent. The most common exceptions that give rise to synchronization problems are the system call service routines (see the section "System Call Handler and Service Routines" in Chapter 10) in which the CPU operates in Kernel Mode to offer a service to a User Mode program. Thus, a data structure

accessed only by an exception usually represents a resource that can be assigned to one or more processes.

Race conditions are avoided through semaphores, because these primitives allow the process to sleep until the resource becomes available. Notice that semaphores work equally well both in uniprocessor and multiprocessor systems.

Kernel preemption does not create problems either. If a process that owns a semaphore is preempted, a new process running on the same CPU could try to get the semaphore. When this occurs, the new process is put to sleep, and eventually the old process will release the semaphore. The only case in which kernel preemption must be explicitly disabled is when accessing per-CPU variables, as explained in the section "Per-CPU Variables" earlier in this chapter.

Protecting a data structure accessed by interrupts

Suppose that a data structure is accessed by only the "top half" of an interrupt handler. We learned in the section "Interrupt Handling" in Chapter 4 that each interrupt handler is serialized with respect to itself—that is, it cannot execute more than once concurrently. Thus, accessing the data structure does not require synchronization primitives.

Things are different, however, if the data structure is accessed by several interrupt handlers. A handler may interrupt another handler, and different interrupt handlers may run concurrently in multiprocessor systems. Without synchronization, the shared data structure might easily become corrupted.

In uniprocessor systems, race conditions must be avoided by disabling interrupts in all critical regions of the interrupt handler. Nothing less will do because no other synchronization primitives accomplish the job. A semaphore can block the process, so it cannot be used in an interrupt handler. A spin lock, on the other hand, can freeze the system: if the handler accessing the data structure is interrupted, it cannot release the lock; therefore, the new interrupt handler keeps waiting on the tight loop of the spin lock.

Multiprocessor systems, as usual, are even more demanding. Race conditions cannot be avoided by simply disabling local interrupts. In fact, even if interrupts are disabled on a CPU, interrupt handlers can still be executed on the other CPUs. The most convenient method to prevent the race conditions is to disable local interrupts (so that other interrupt handlers running on the same CPU won't interfere) *and* to acquire a spin lock or a read/write spin lock that protects the data structure. Notice that these additional spin locks cannot freeze the system because even if an interrupt handler finds the lock closed, eventually the interrupt handler on the other CPU that owns the lock will release it.

The Linux kernel uses several macros that couple the enabling and disabling of local interrupts with spin lock handling. Table 5-9 describes all of them. In uniprocessor systems, these macros just enable or disable local interrupts and kernel preemption.

Table 5-9. Interrupt-aware spin lock macros

Macro	Description
spin_lock_irq(l)	local_irq_disable(); spin_lock(l)
spin_unlock_irq(l)	spin_unlock(l); local_irq_enable()
spin_lock_bh(l)	local_bh_disable(); spin_lock(l)
spin_unlock_bh(l)	spin_unlock(l); local_bh_enable()
spin_lock_irqsave(l,f)	local_irq_save(f); spin_lock(l)
spin_unlock_irqrestore(l,f)	spin_unlock(l); local_irq_restore(f)
read_lock_irq(l)	local_irq_disable( ); read_lock(l)
read_unlock_irq(l)	read_unlock(l); local_irq_enable( )
read_lock_bh(l)	local_bh_disable( ); read_lock(l)
read_unlock_bh(l)	read_unlock(l); local_bh_enable( )
write_lock_irq(l)	local_irq_disable(); write_lock(l)
write_unlock_irq(l)	write_unlock(l); local_irq_enable( )
write_lock_bh(l)	local_bh_disable(); write_lock(l)
write_unlock_bh(l)	write_unlock(l); local_bh_enable( )
read_lock_irqsave(l,f)	local_irq_save(f); read_lock(l)
read_unlock_irqrestore(l,f)	read_unlock(l); local_irq_restore(f)
write_lock_irqsave(l,f)	local_irq_save(f); write_lock(l)
write_unlock_irqrestore(l,f)	write_unlock(l); local_irq_restore(f)
read_seqbegin_irqsave(l,f)	local_irq_save(f); read_seqbegin(l)
read_seqretry_irqrestore(l,v,f)	read_seqretry(l,v); local_irq_restore(f)
write_seqlock_irqsave(l,f)	local_irq_save(f); write_seqlock(l)
write_sequnlock_irqrestore(l,f)	write_sequnlock(l); local_irq_restore(f)
write_seqlock_irq(l)	local_irq_disable(); write_seqlock(l)
write_sequnlock_irq(l)	write_sequnlock(l); local_irq_enable()
write_seqlock_bh(l)	local_bh_disable(); write_seqlock(l);
write_sequnlock_bh(l)	write_sequnlock(l); local_bh_enable()

Protecting a data structure accessed by deferrable functions

What kind of protection is required for a data structure accessed only by deferrable functions? Well, it mostly depends on the kind of deferrable function. In the section "Softirqs and Tasklets" in Chapter 4, we explained that softirqs and tasklets essentially differ in their degree of concurrency.

First of all, no race condition may exist in uniprocessor systems. This is because execution of deferrable functions is always serialized on a CPU—that is, a deferrable function cannot be interrupted by another deferrable function. Therefore, no synchronization primitive is ever required.

Conversely, in multiprocessor systems, race conditions do exist because several deferrable functions may run concurrently. Table 5-10 lists all possible cases.

Table 5-10. Protection required by data structures accessed by deferrable functions in SMP

Deferrable functions accessing the data structure	Protection
Softirqs	Spin lock
One tasklet	None
Many tasklets	Spin lock

A data structure accessed by a softirq must always be protected, usually by means of a spin lock, because the same softirq may run concurrently on two or more CPUs. Conversely, a data structure accessed by just one kind of tasklet need not be protected, because tasklets of the same kind cannot run concurrently. However, if the data structure is accessed by several kinds of tasklets, then it must be protected.

Protecting a data structure accessed by exceptions and interrupts

Let's consider now a data structure that is accessed both by exceptions (for instance, system call service routines) and interrupt handlers.

On uniprocessor systems, race condition prevention is quite simple, because interrupt handlers are not reentrant and cannot be interrupted by exceptions. As long as the kernel accesses the data structure with local interrupts disabled, the kernel cannot be interrupted when accessing the data structure. However, if the data structure is accessed by just one kind of interrupt handler, the interrupt handler can freely access the data structure without disabling local interrupts.

On multiprocessor systems, we have to take care of concurrent executions of exceptions and interrupts on other CPUs. Local interrupt disabling must be coupled with a spin lock, which forces the concurrent kernel control paths to wait until the handler accessing the data structure finishes its work.

Sometimes it might be preferable to replace the spin lock with a semaphore. Because interrupt handlers cannot be suspended, they must acquire the semaphore using a tight loop and the `down_trylock()` function; for them, the semaphore acts essentially as a spin lock. System call service routines, on the other hand, may suspend the calling processes when the semaphore is busy. For most system calls, this is the expected behavior. In this case, semaphores are preferable to spin locks, because they lead to a higher degree of concurrency of the system.

Protecting a data structure accessed by exceptions and deferrable functions

A data structure accessed both by exception handlers and deferrable functions can be treated like a data structure accessed by exception and interrupt handlers. In fact, deferrable functions are essentially activated by interrupt occurrences, and no exception can be raised while a deferrable function is running. Coupling local interrupt disabling with a spin lock is therefore sufficient.

Actually, this is much more than sufficient: the exception handler can simply disable deferrable functions instead of local interrupts by using the `local_bh_disable()` macro (see the section "Softirqs" in Chapter 4). Disabling only the deferrable functions is preferable to disabling interrupts, because interrupts continue to be serviced by the CPU. Execution of deferrable functions on each CPU is serialized, so no race condition exists.

As usual, in multiprocessor systems, spin locks are required to ensure that the data structure is accessed at any time by just one kernel control.

Protecting a data structure accessed by interrupts and deferrable functions

This case is similar to that of a data structure accessed by interrupt and exception handlers. An interrupt might be raised while a deferrable function is running, but no deferrable function can stop an interrupt handler. Therefore, race conditions must be avoided by disabling local interrupts during the deferrable function. However, an interrupt handler can freely touch the data structure accessed by the deferrable function without disabling interrupts, provided that no other interrupt handler accesses that data structure.

Again, in multiprocessor systems, a spin lock is always required to forbid concurrent accesses to the data structure on several CPUs.

Protecting a data structure accessed by exceptions, interrupts, and deferrable functions

Similarly to previous cases, disabling local interrupts and acquiring a spin lock is almost always necessary to avoid race conditions. Notice that there is no need to explicitly disable deferrable functions, because they are essentially activated when terminating the execution of interrupt handlers; disabling local interrupts is therefore sufficient.

Examples of Race Condition Prevention

Kernel developers are expected to identify and solve the synchronization problems raised by interleaving kernel control paths. However, avoiding race conditions is a hard task because it requires a clear understanding of how the various components of the kernel interact. To give a feeling of what's really inside the kernel code, let's mention a few typical usages of the synchronization primitives defined in this chapter.

Reference Counters

Reference counters are widely used inside the kernel to avoid race conditions due to the concurrent allocation and releasing of a resource. A *reference counter* is just an `atomic_t` counter associated with a specific resource such as a memory page, a module, or a file. The counter is atomically increased when a kernel control path starts using the resource, and it is decreased when a kernel control path finishes using the resource. When the reference counter becomes zero, the resource is not being used, and it can be released if necessary.

The Big Kernel Lock

In earlier Linux kernel versions, a *big kernel lock* (also known as *global kernel lock*, or *BKL*) was widely used. In Linux 2.0, this lock was a relatively crude spin lock, ensuring that only one processor at a time could run in Kernel Mode. The 2.2 and 2.4 kernels were considerably more flexible and no longer relied on a single spin lock; rather, a large number of kernel data structures were protected by many different spin locks. In Linux kernel version 2.6, the big kernel lock is used to protect old code (mostly functions related to the VFS and to several filesystems).

Starting from kernel version 2.6.11, the big kernel lock is implemented by a semaphore named `kernel_sem` (in earlier 2.6 versions, the big kernel lock was implemented by means of a spin lock). The big kernel lock is slightly more sophisticated than a simple semaphore, however.

Every process descriptor includes a `lock_depth` field, which allows the same process to acquire the big kernel lock several times. Therefore, two consecutive requests for it will not hang the processor (as for normal locks). If the process has not acquired the lock, the field has the value –1; otherwise, the field value plus 1 specifies how many times the lock has been taken. The `lock_depth` field is crucial for allowing interrupt handlers, exception handlers, and deferrable functions to take the big kernel lock: without it, every asynchronous function that tries to get the big kernel lock could generate a deadlock if the current process already owns the lock.

The `lock_kernel()` and `unlock_kernel()` functions are used to get and release the big kernel lock. The former function is equivalent to:

```
depth = current->lock_depth + 1;
if (depth == 0)
    down(&kernel_sem);
current->lock_depth = depth;
```

while the latter is equivalent to:

```
if (--current->lock_depth < 0)
    up(&kernel_sem);
```

Notice that the `if` statements of the `lock_kernel()` and `unlock_kernel()` functions need not be executed atomically because `lock_depth` is not a global variable—each

CPU addresses a field of its own current process descriptor. Local interrupts inside the `if` statements do not induce race conditions either. Even if the new kernel control path invokes `lock_kernel()`, it must release the big kernel lock before terminating.

Surprisingly enough, a process holding the big kernel lock is allowed to invoke `schedule()`, thus relinquishing the CPU. The `schedule()` function, however, checks the `lock_depth` field of the process being replaced and, if its value is zero or positive, automatically releases the `kernel_sem` semaphore (see the section "The schedule() Function" in Chapter 7). Thus, no process that explicitly invokes `schedule()` can keep the big kernel lock across the process switch. The `schedule()` function, however, will reacquire the big kernel lock for the process when it will be selected again for execution.

Things are different, however, if a process that holds the big kernel lock is preempted by another process. Up to kernel version 2.6.10 this case could not occur, because acquiring a spin lock automatically disables kernel preemption. The current implementation of the big kernel lock, however, is based on a semaphore, and acquiring it does not automatically disable kernel preemption. Actually, allowing kernel preemption inside critical regions protected by the big kernel lock has been the main reason for changing its implementation. This, in turn, has beneficial effects on the response time of the system.

When a process holding the big kernel lock is preempted, `schedule()` must not release the semaphore because the process executing the code in the critical region has not voluntarily triggered a process switch, thus if the big kernel lock would be released, another process might take it and corrupt the data structures accessed by the preempted process.

To avoid the preempted process losing the big kernel lock, the `preempt_schedule_irq()` function temporarily sets the `lock_depth` field of the process to -1 (see the section "Returning from Interrupts and Exceptions" in Chapter 4). Looking at the value of this field, `schedule()` assumes that the process being replaced does not hold the `kernel_sem` semaphore and thus does not release it. As a result, the `kernel_sem` semaphore continues to be owned by the preempted process. Once this process is selected again by the scheduler, the `preempt_schedule_irq()` function restores the original value of the `lock_depth` field and lets the process resume execution in the critical section protected by the big kernel lock.

Memory Descriptor Read/Write Semaphore

Each memory descriptor of type `mm_struct` includes its own semaphore in the `mmap_sem` field (see the section "The Memory Descriptor" in Chapter 9). The semaphore protects the descriptor against race conditions that could arise because a memory descriptor can be shared among several lightweight processes.

For instance, let's suppose that the kernel must create or extend a memory region for some process; to do this, it invokes the `do_mmap()` function, which allocates a new `vm_area_struct` data structure. In doing so, the current process could be suspended if no free memory is available, and another process sharing the same memory descriptor could run. Without the semaphore, every operation of the second process that requires access to the memory descriptor (for instance, a Page Fault due to a Copy on Write) could lead to severe data corruption.

The semaphore is implemented as a read/write semaphore, because some kernel functions, such as the Page Fault exception handler (see the section "Page Fault Exception Handler" in Chapter 9), need only to scan the memory descriptors.

Slab Cache List Semaphore

The list of slab cache descriptors (see the section "Cache Descriptor" in Chapter 8) is protected by the `cache_chain_sem` semaphore, which grants an exclusive right to access and modify the list.

A race condition is possible when `kmem_cache_create()` adds a new element in the list, while `kmem_cache_shrink()` and `kmem_cache_reap()` sequentially scan the list. However, these functions are never invoked while handling an interrupt, and they can never block while accessing the list. The semaphore plays an active role both in multiprocessor systems and in uniprocessor systems with kernel preemption supported.

Inode Semaphore

As we'll see in "Inode Objects" in Chapter 12, Linux stores the information on a disk file in a memory object called an inode. The corresponding data structure includes its own semaphore in the `i_sem` field.

A huge number of race conditions can occur during filesystem handling. Indeed, each file on disk is a resource held in common for all users, because all processes may (potentially) access the file content, change its name or location, destroy or duplicate it, and so on. For example, let's suppose that a process lists the files contained in some directory. Each disk operation is potentially blocking, and therefore even in uniprocessor systems, other processes could access the same directory and modify its content while the first process is in the middle of the listing operation. Or, again, two different processes could modify the same directory at the same time. All these race conditions are avoided by protecting the directory file with the inode semaphore.

Whenever a program uses two or more semaphores, the potential for deadlock is present, because two different paths could end up waiting for each other to release a semaphore. Generally speaking, Linux has few problems with deadlocks on semaphore requests, because each kernel control path usually needs to acquire just one semaphore at a time. However, in some cases, the kernel must get two or more locks.

Inode semaphores are prone to this scenario; for instance, this occurs in the service routine in the `rename( )` system call. In this case, two different inodes are involved in the operation, so both semaphores must be taken. To avoid such deadlocks, semaphore requests are performed in predefined address order.

CHAPTER 7

Process Scheduling

Like every time sharing system, Linux achieves the magical effect of an apparent simultaneous execution of multiple processes by switching from one process to another in a very short time frame. Process switching itself was discussed in Chapter 3; this chapter deals with *scheduling*, which is concerned with when to switch and which process to choose.

The chapter consists of three parts. The section "Scheduling Policy" introduces the choices made by Linux in the abstract to schedule processes. The section "The Scheduling Algorithm" discusses the data structures used to implement scheduling and the corresponding algorithm. Finally, the section "System Calls Related to Scheduling" describes the system calls that affect process scheduling.

To simplify the description, we refer as usual to the 80×86 architecture; in particular, we assume that the system uses the Uniform Memory Access model, and that the system tick is set to 1 ms.

Scheduling Policy

The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and so on. The set of rules used to determine when and how to select a new process to run is called *scheduling policy*.

Linux scheduling is based on the *time sharing* technique: several processes run in "time multiplexing" because the CPU time is divided into *slices*, one for each runnable process.* Of course, a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or *quantum* expires, a

* Recall that stopped and suspended processes cannot be selected by the scheduling algorithm to run on a CPU.

process switch may take place. Time sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs to ensure CPU time sharing.

The scheduling policy is also based on ranking processes according to their priority. Complicated algorithms are sometimes used to derive the current priority of a process, but the end result is the same: each process is associated with a value that tells the scheduler how appropriate it is to let the process run on a CPU.

In Linux, process priority is dynamic. The scheduler keeps track of what processes are doing and adjusts their priorities periodically; in this way, processes that have been denied the use of a CPU for a long time interval are boosted by dynamically increasing their priority. Correspondingly, processes running for a long time are penalized by decreasing their priority.

When speaking about scheduling, processes are traditionally classified as *I/O-bound* or *CPU-bound*. The former make heavy use of I/O devices and spend much time waiting for I/O operations to complete; the latter carry on number-crunching applications that require a lot of CPU time.

An alternative classification distinguishes three classes of processes:

Interactive processes
: These interact constantly with their users, and therefore spend a lot of time waiting for keypresses and mouse operations. When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive. Typically, the average delay must fall between 50 and 150 milliseconds. The variance of such delay must also be bounded, or the user will find the system to be erratic. Typical interactive programs are command shells, text editors, and graphical applications.

Batch processes
: These do not need user interaction, and hence they often run in the background. Because such processes do not need to be very responsive, they are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines, and scientific computations.

Real-time processes
: These have very stringent scheduling requirements. Such processes should never be blocked by lower-priority processes and should have a short guaranteed response time with a minimum variance. Typical real-time programs are video and sound applications, robot controllers, and programs that collect data from physical sensors.

The two classifications we just offered are somewhat independent. For instance, a batch process can be either I/O-bound (e.g., a database server) or CPU-bound (e.g., an image-rendering program). While real-time programs are explicitly recognized as such by the scheduling algorithm in Linux, there is no easy way to distinguish

between interactive and batch programs. The Linux 2.6 scheduler implements a sophisticated heuristic algorithm based on the past behavior of the processes to decide whether a given process should be considered as interactive or batch. Of course, the scheduler tends to favor interactive processes over batch ones.

Programmers may change the scheduling priorities by means of the system calls illustrated in Table 7-1. More details are given in the section "System Calls Related to Scheduling."

Table 7-1. System calls related to scheduling

System call	Description
`nice()`	Change the static priority of a conventional process
`getpriority()`	Get the maximum static priority of a group of conventional processes
`setpriority()`	Set the static priority of a group of conventional processes
`sched_getscheduler()`	Get the scheduling policy of a process
`sched_setscheduler()`	Set the scheduling policy and the real-time priority of a process
`sched_getparam()`	Get the real-time priority of a process
`sched_setparam()`	Set the real-time priority of a process
`sched_yield()`	Relinquish the processor voluntarily without blocking
`sched_get_priority_min()`	Get the minimum real-time priority value for a policy
`sched_get_priority_max()`	Get the maximum real-time priority value for a policy
`sched_rr_get_interval()`	Get the time quantum value for the Round Robin policy
`sched_setaffinity()`	Set the CPU affinity mask of a process
`sched_getaffinity()`	Get the CPU affinity mask of a process

Process Preemption

As mentioned in the first chapter, Linux processes are *preemptable*. When a process enters the `TASK_RUNNING` state, the kernel checks whether its dynamic priority is greater than the priority of the currently running process. If it is, the execution of `current` is interrupted and the scheduler is invoked to select another process to run (usually the process that just became runnable). Of course, a process also may be preempted when its time quantum expires. When this occurs, the `TIF_NEED_RESCHED` flag in the `thread_info` structure of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.

For instance, let's consider a scenario in which only two programs—a text editor and a compiler—are being executed. The text editor is an interactive program, so it has a higher dynamic priority than the compiler. Nevertheless, it is often suspended, because the user alternates between pauses for think time and data entry; moreover, the average delay between two keypresses is relatively long. However, as soon as the user presses a key, an interrupt is raised and the kernel wakes up the text editor process. The kernel also determines that the dynamic priority of the editor is higher than

the priority of current, the currently running process (the compiler), so it sets the TIF_NEED_RESCHED flag of this process, thus forcing the scheduler to be activated when the kernel finishes handling the interrupt. The scheduler selects the editor and performs a process switch; as a result, the execution of the editor is resumed very quickly and the character typed by the user is echoed to the screen. When the character has been processed, the text editor process suspends itself waiting for another keypress and the compiler process can resume its execution.

Be aware that a preempted process is not suspended, because it remains in the TASK_RUNNING state; it simply no longer uses the CPU. Moreover, remember that the Linux 2.6 kernel is preemptive, which means that a process can be preempted either when executing in Kernel or in User Mode; we discussed in depth this feature in the section "Kernel Preemption" in Chapter 5.

How Long Must a Quantum Last?

The quantum duration is critical for system performance: it should be neither too long nor too short.

If the average quantum duration is too short, the system overhead caused by process switches becomes excessively high. For instance, suppose that a process switch requires 5 milliseconds; if the quantum is also set to 5 milliseconds, then at least 50 percent of the CPU cycles will be dedicated to process switching.*

If the average quantum duration is too long, processes no longer appear to be executed concurrently. For instance, let's suppose that the quantum is set to five seconds; each runnable process makes progress for about five seconds, but then it stops for a very long time (typically, five seconds times the number of runnable processes).

It is often believed that a long quantum duration degrades the response time of interactive applications. This is usually false. As described in the section "Process Preemption" earlier in this chapter, interactive processes have a relatively high priority, so they quickly preempt the batch processes, no matter how long the quantum duration is.

In some cases, however, a very long quantum duration degrades the responsiveness of the system. For instance, suppose two users concurrently enter two commands at the respective shell prompts; one command starts a CPU-bound process, while the other launches an interactive application. Both shells fork a new process and delegate the execution of the user's command to it; moreover, suppose such new processes have the same initial priority (Linux does not know in advance if a program to be executed is batch or interactive). Now if the scheduler selects the CPU-bound process to run

* Actually, things could be much worse than this; for example, if the time required for the process switch is counted in the process quantum, all CPU time is devoted to the process switch and no process can progress toward its termination.

first, the other process could wait for a whole time quantum before starting its execution. Therefore, if the quantum duration is long, the system could appear to be unresponsive to the user that launched the interactive application.

The choice of the average quantum duration is always a compromise. The rule of thumb adopted by Linux is choose a duration as long as possible, while keeping good system response time.

The Scheduling Algorithm

The scheduling algorithm used in earlier versions of Linux was quite simple and straightforward: at every process switch the kernel scanned the list of runnable processes, computed their priorities, and selected the "best" process to run. The main drawback of that algorithm is that the time spent in choosing the best process depends on the number of runnable processes; therefore, the algorithm is too costly—that is, it spends too much time—in high-end systems running thousands of processes.

The scheduling algorithm of Linux 2.6 is much more sophisticated. By design, it scales well with the number of runnable processes, because it selects the process to run in constant time, independently of the number of runnable processes. It also scales well with the number of processors because each CPU has its own queue of runnable processes. Furthermore, the new algorithm does a better job of distinguishing interactive processes and batch processes. As a consequence, users of heavily loaded systems feel that interactive applications are much more responsive in Linux 2.6 than in earlier versions.

The scheduler always succeeds in finding a process to be executed; in fact, there is always at least one runnable process: the *swapper* process, which has PID 0 and executes only when the CPU cannot execute other processes. As mentioned in Chapter 3, every CPU of a multiprocessor system has its own *swapper* process with PID equal to 0.

Every Linux process is always scheduled according to one of the following *scheduling classes*:

`SCHED_FIFO`
: A First-In, First-Out real-time process. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list. If no other higher-priority real-time process is runnable, the process continues to use the CPU as long as it wishes, even if other real-time processes that have the same priority are runnable.

`SCHED_RR`
: A Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list. This policy ensures a fair assignment of CPU time to all `SCHED_RR` real-time processes that have the same priority.

SCHED_NORMAL
: A conventional, time-shared process.

The scheduling algorithm behaves quite differently depending on whether the process is conventional or real-time.

Scheduling of Conventional Processes

Every conventional process has its own *static priority*, which is a value used by the scheduler to rate the process with respect to the other conventional processes in the system. The kernel represents the static priority of a conventional process with a number ranging from 100 (highest priority) to 139 (lowest priority); notice that static priority decreases as the values increase.

A new process always inherits the static priority of its parent. However, a user can change the static priority of the processes that he owns by passing some "nice values" to the `nice()` and `setpriority()` system calls (see the section "System Calls Related to Scheduling" later in this chapter).

Base time quantum

The static priority essentially determines the *base time quantum* of a process, that is, the time quantum duration assigned to the process when it has exhausted its previous time quantum. Static priority and base time quantum are related by the following formula:

$$\underset{\text{(in milliseconds)}}{\textit{base time quantum}} = \begin{cases} (140 - \textit{static priority}) \times 20 & \text{if } \textit{static priority} < 120 \\ (140 - \textit{static priority}) \times 5 & \text{if } \textit{static priority} \geq 120 \end{cases} \quad (1)$$

As you see, the higher the static priority (i.e., the lower its numerical value), the longer the base time quantum. As a consequence, higher priority processes usually get longer slices of CPU time with respect to lower priority processes. Table 7-2 shows the static priority, the base time quantum values, and the corresponding nice values for a conventional process having highest static priority, default static priority, and lowest static priority. (The table also lists the values of the interactive delta and of the sleep time threshold, which are explained later in this chapter.)

Table 7-2. Typical priority values for a conventional process

Description	Static priority	Nice value	Base time quantum	Interactivedelta	Sleep time threshold
Highest static priority	100	–20	800 ms	–3	299 ms
High static priority	110	-10	600 ms	-1	499 ms
Default static priority	120	0	100 ms	+2	799 ms
Low static priority	130	+10	50 ms	+4	999 ms
Lowest static priority	139	+19	5 ms	+6	1199 ms

Dynamic priority and average sleep time

Besides a static priority, a conventional process also has a *dynamic priority*, which is a value ranging from 100 (highest priority) to 139 (lowest priority). The dynamic priority is the number actually looked up by the scheduler when selecting the new process to run. It is related to the static priority by the following empirical formula:

$$dynamic\ priority = \max(100, \min(static\ priority - bonus + 5, 139)) \qquad (2)$$

The *bonus* is a value ranging from 0 to 10; a value less than 5 represents a penalty that lowers the dynamic priority, while a value greater than 5 is a premium that raises the dynamic priority. The value of the bonus, in turn, depends on the past history of the process; more precisely, it is related to the *average sleep time* of the process.

Roughly, the average sleep time is the average number of nanoseconds that the process spent while sleeping. Be warned, however, that this is not an average operation on the elapsed time. For instance, sleeping in `TASK_INTERRUPTIBLE` state contributes to the average sleep time in a different way from sleeping in `TASK_UNINTERRUPTIBLE` state. Moreover, the average sleep time decreases while a process is running. Finally, the average sleep time can never become larger than 1 second.

The correspondence between average sleep times and bonus values is shown in Table 7-3. (The table lists also the corresponding granularity of the time slice, which will be discussed later.)

Table 7-3. Average sleep times, bonus values, and time slice granularity

Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10

The average sleep time is also used by the scheduler to determine whether a given process should be considered interactive or batch. More precisely, a process is considered "interactive" if it satisfies the following formula:

$$dynamic\ priority \leq 3 \times static\ priority / 4 + 28 \qquad (3)$$

which is equivalent to the following:

bonus - 5 ≥ *static priority* / 4 – 28

The expression *static priority* / 4 – 28 is called the *interactive delta*; some typical values of this term are listed in Table 7-2. It should be noted that it is far easier for high priority than for low priority processes to become interactive. For instance, a process having highest static priority (100) is considered interactive when its bonus value exceeds 2, that is, when its average sleep time exceeds 200 ms. Conversely, a process having lowest static priority (139) is never considered as interactive, because the bonus value is always smaller than the value 11 required to reach an interactive delta equal to 6. A process having default static priority (120) becomes interactive as soon as its average sleep time exceeds 700 ms.

Active and expired processes

Even if conventional processes having higher static priorities get larger slices of the CPU time, they should not completely lock out the processes having lower static priority. To avoid process starvation, when a process finishes its time quantum, it can be replaced by a lower priority process whose time quantum has not yet been exhausted. To implement this mechanism, the scheduler keeps two disjoint sets of runnable processes:

Active processes
: These runnable processes have not yet exhausted their time quantum and are thus allowed to run.

Expired processes
: These runnable processes have exhausted their time quantum and are thus forbidden to run until all active processes expire.

However, the general schema is slightly more complicated than this, because the scheduler tries to boost the performance of interactive processes. An active batch process that finishes its time quantum always becomes expired. An active interactive process that finishes its time quantum usually remains active: the scheduler refills its time quantum and leaves it in the set of active processes. However, the scheduler moves an interactive process that finished its time quantum into the set of expired processes if the eldest expired process has already waited for a long time, or if an expired process has higher static priority (lower value) than the interactive process. As a consequence, the set of active processes will eventually become empty and the expired processes will have a chance to run.

Scheduling of Real-Time Processes

Every real-time process is associated with a *real-time priority*, which is a value ranging from 1 (highest priority) to 99 (lowest priority). The scheduler always favors a higher priority runnable process over a lower priority one; in other words, a real-time

process inhibits the execution of every lower-priority process while it remains runnable. Contrary to conventional processes, real-time processes are always considered active (see the previous section). The user can change the real-time priority of a process by means of the `sched_setparam()` and `sched_setscheduler()` system calls (see the section "System Calls Related to Scheduling" later in this chapter).

If several real-time runnable processes have the same highest priority, the scheduler chooses the process that occurs first in the corresponding list of the local CPU's runqueue (see the section "The lists of TASK_RUNNING processes" in Chapter 3).

A real-time process is replaced by another process only when one of the following events occurs:

- The process is preempted by another process having higher real-time priority.
- The process performs a blocking operation, and it is put to sleep (in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`).
- The process is stopped (in state `TASK_STOPPED` or `TASK_TRACED`), or it is killed (in state `EXIT_ZOMBIE` or `EXIT_DEAD`).
- The process voluntarily relinquishes the CPU by invoking the `sched_yield()` system call (see the section "System Calls Related to Scheduling" later in this chapter).
- The process is Round Robin real-time (`SCHED_RR`), and it has exhausted its time quantum.

The `nice()` and `setpriority()` system calls, when applied to a Round Robin real-time process, do not change the real-time priority but rather the duration of the base time quantum. In fact, the duration of the base time quantum of Round Robin real-time processes does not depend on the real-time priority, but rather on the static priority of the process, according to the formula (1) in the earlier section "Scheduling of Conventional Processes."

Data Structures Used by the Scheduler

Recall from the section "Identifying a Process" in Chapter 3 that the process list links all process descriptors, while the runqueue lists link the process descriptors of all runnable processes—that is, of those in a `TASK_RUNNING` state—except the *swapper* process (idle process).

The runqueue Data Structure

The `runqueue` data structure is the most important data structure of the Linux 2.6 scheduler. Each CPU in the system has its own runqueue; all `runqueue` structures are stored in the `runqueues` per-CPU variable (see the section "Per-CPU Variables" in Chapter 5). The `this_rq()` macro yields the address of the runqueue of the local CPU, while the `cpu_rq(n)` macro yields the address of the runqueue of the CPU having index `n`.

Table 7-4 lists the fields included in the runqueue data structure; we will discuss most of them in the following sections of the chapter.

Table 7-4. The fields of the runqueue structure

Type	Name	Description
spinlock_t	lock	Spin lock protecting the lists of processes
unsigned long	nr_running	Number of runnable processes in the runqueue lists
unsigned long	cpu_load	CPU load factor based on the average number of processes in the runqueue
unsigned long	nr_switches	Number of process switches performed by the CPU
unsigned long	nr_uninterruptible	Number of processes that were previously in the runqueue lists and are now sleeping in TASK_UNINTERRUPTIBLE state (only the sum of these fields across all runqueues is meaningful)
unsigned long	expired_timestamp	Insertion time of the eldest process in the expired lists
unsigned long long	timestamp_last_tick	Timestamp value of the last timer interrupt
task_t *	curr	Process descriptor pointer of the currently running process (same as current for the local CPU)
task_t *	idle	Process descriptor pointer of the *swapper* process for this CPU
struct mm_struct *	prev_mm	Used during a process switch to store the address of the memory descriptor of the process being replaced
prio_array_t *	active	Pointer to the lists of active processes
prio_array_t *	expired	Pointer to the lists of expired processes
prio_array_t [2]	arrays	The two sets of active and expired processes
int	best_expired_prio	The best static priority (lowest value) among the expired processes
atomic_t	nr_iowait	Number of processes that were previously in the runqueue lists and are now waiting for a disk I/O operation to complete
struct sched_domain *	sd	Points to the base scheduling domain of this CPU (see the section "Scheduling Domains" later in this chapter)
int	active_balance	Flag set if some process shall be *migrated* from this runqueue to another (runqueue balancing)
int	push_cpu	Not used
task_t *	migration_thread	Process descriptor pointer of the *migration* kernel thread
struct list_head	migration_queue	List of processes to be removed from the runqueue

The most important fields of the runqueue data structure are those related to the lists of runnable processes. Every runnable process in the system belongs to one, and just one, runqueue. As long as a runnable process remains in the same runqueue, it can

be executed only by the CPU owning that runqueue. However, as we'll see later, runnable processes may migrate from one runqueue to another.

The arrays field of the runqueue is an array consisting of two prio_array_t structures. Each data structure represents a set of runnable processes, and includes 140 doubly linked list heads (one list for each possible process priority), a priority bitmap, and a counter of the processes included in the set (see Table 3-2 in the section Chapter 3).

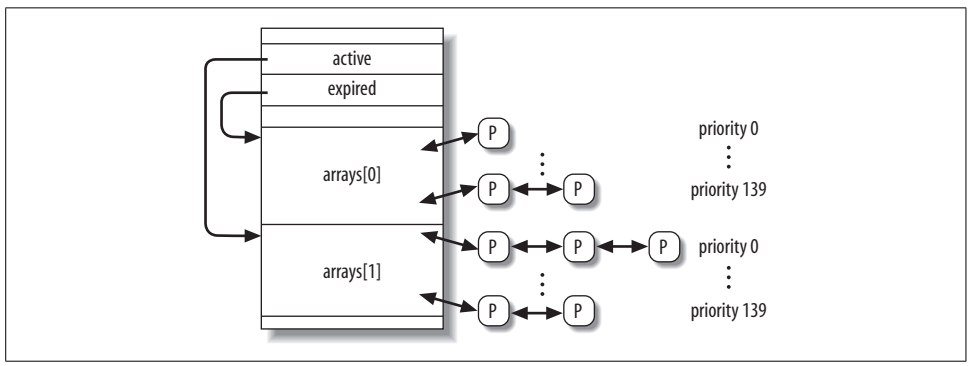


Figure 7-1. The runqueue structure and the two sets of runnable processes

As shown in Figure 7-1, the active field of the runqueue structure points to one of the two prio_array_t data structures in arrays: the corresponding set of runnable processes includes the active processes. Conversely, the expired field points to the other prio_array_t data structure in arrays: the corresponding set of runnable processes includes the expired processes.

Periodically, the role of the two data structures in arrays changes: the active processes suddenly become the expired processes, and the expired processes become the active ones. To achieve this change, the scheduler simply exchanges the contents of the active and expired fields of the runqueue.

Process Descriptor

Each process descriptor includes several fields related to scheduling; they are listed in Table 7-5.

Table 7-5. Fields of the process descriptor related to the scheduler

Type	Name	Description
unsigned long	thread_info->flags	Stores the TIF_NEED_RESCHED flag, which is set if the scheduler must be invoked (see the section "Returning from Interrupts and Exceptions" in Chapter 4)
unsigned int	thread_info->cpu	Logical number of the CPU owning the runqueue to which the runnable process belongs

Table 7-5. Fields of the process descriptor related to the scheduler (continued)

Type	Name	Description
unsigned long	state	The current state of the process (see the section "Process State" in Chapter 3)
int	prio	Dynamic priority of the process
int	static_prio	Static priority of the process
struct list_head	run_list	Pointers to the next and previous elements in the runqueue list to which the process belongs
prio_array_t *	array	Pointer to the runqueue's prio_array_t set that includes the process
unsigned long	sleep_avg	Average sleep time of the process
unsigned long long	timestamp	Time of last insertion of the process in the runqueue, or time of last process switch involving the process
unsigned long long	last_ran	Time of last process switch that replaced the process
int	activated	Condition code used when the process is awakened
unsigned long	policy	The scheduling class of the process (SCHED_NORMAL, SCHED_RR, or SCHED_FIFO)
cpumask_t	cpus_allowed	Bit mask of the CPUs that can execute the process
unsigned int	time_slice	Ticks left in the time quantum of the process
unsigned int	first_time_slice	Flag set to 1 if the process never exhausted its time quantum
unsigned long	rt_priority	Real-time priority of the process

When a new process is created, sched_fork(), invoked by copy_process(), sets the time_slice field of both current (the parent) and p (the child) processes in the following way:

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```

In other words, the number of ticks left to the parent is split in two halves: one for the parent and one for the child. This is done to prevent users from getting an unlimited amount of CPU time by using the following method: the parent process creates a child process that runs the same code and then kills itself; by properly adjusting the creation rate, the child process would always get a fresh quantum before the quantum of its parent expires. This programming trick does not work because the kernel does not reward forks. Similarly, a user cannot hog an unfair share of the processor by starting several background processes in a shell or by opening a lot of windows on a graphical desktop. More generally speaking, a process cannot hog resources (unless it has privileges to give itself a real-time policy) by forking multiple descendents.

If the parent had just one tick left in its time slice, the splitting operation forces current->time_slice to 0, thus exhausting the quantum of the parent. In this case, copy_process() sets current->time_slice back to 1, then invokes scheduler_tick() to decrease the field (see the following section).

The copy_process() function also initializes a few other fields of the child's process descriptor related to scheduling:

```
p->first_time_slice = 1;
p->timestamp = sched_clock();
```

The first_time_slice flag is set to 1, because the child has never exhausted its time quantum (if a process terminates or executes a new program during its first time slice, the parent process is rewarded with the remaining time slice of the child). The timestamp field is initialized with a timestamp value produced by sched_clock(): essentially, this function returns the contents of the 64-bit TSC register (see the section "Time Stamp Counter (TSC)" in Chapter 6) converted to nanoseconds.

Functions Used by the Scheduler

The scheduler relies on several functions in order to do its work; the most important are:

scheduler_tick()
: Keeps the time_slice counter of current up-to-date

try_to_wake_up()
: Awakens a sleeping process

recalc_task_prio()
: Updates the dynamic priority of a process

schedule()
: Selects a new process to be executed

load_balance()
: Keeps the runqueues of a multiprocessor system balanced

The scheduler_tick( ) Function

We have already explained in the section "Updating Local CPU Statistics" in Chapter 6 how scheduler_tick() is invoked once every tick to perform some operations related to scheduling. It executes the following main steps:

1. Stores in the timestamp_last_tick field of the local runqueue the current value of the TSC converted to nanoseconds; this timestamp is obtained from the sched_clock() function (see the previous section).
2. Checks whether the current process is the *swapper* process of the local CPU. If so, it performs the following substeps:
   a. If the local runqueue includes another runnable process besides *swapper*, it sets the TIF_NEED_RESCHED flag of the current process to force rescheduling. As we'll see in the section "The schedule( ) Function" later in this chapter, if the kernel supports the hyper-threading technology (see the section "Runqueue Balancing in Multiprocessor Systems" later in this chapter), a logical

CPU might be idle even if there are runnable processes in its runqueue, as long as those processes have significantly lower priorities than the priority of a process already executing on another logical CPU associated with the same physical CPU.

b. Jumps to step 7 (there is no need to update the time slice counter of the *swapper* process).

3. Checks whether `current->array` points to the active list of the local runqueue. If not, the process has expired its time quantum, but it has not yet been replaced: sets the `TIF_NEED_RESCHED` flag to force rescheduling, and jumps to step 7.
4. Acquires the `this_rq()->lock` spin lock.
5. Decreases the time slice counter of the current process, and checks whether the quantum is exhausted. The operations performed by the function are quite different according to the scheduling class of the process; we will discuss them in a moment.
6. Releases the `this_rq()->lock` spin lock.
7. Invokes the `rebalance_tick()` function, which should ensure that the runqueues of the various CPUs contain approximately the same number of runnable processes. We will discuss runqueue balancing in the later section "Runqueue Balancing in Multiprocessor Systems."

Updating the time slice of a real-time process

If the current process is a FIFO real-time process, `scheduler_tick()` has nothing to do. In this case, in fact, `current` cannot be preempted by lower or equal priority processes, thus it does not make sense to keep its time slice counter up-to-date.

If `current` is a Round Robin real-time process, `scheduler_tick()` decreases its time slice counter and checks whether the quantum is exhausted:

```
if (current->policy == SCHED_RR && !--current->time_slice) {
    current->time_slice = task_timeslice(current);
    current->first_time_slice = 0;
    set_tsk_need_resched(current);
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
                  this_rq()->active->queue+current->prio);
}
```

If the function determines that the time quantum is effectively exhausted, it performs a few operations aimed to ensure that `current` will be preempted, if necessary, as soon as possible.

The first operation consists of refilling the time slice counter of the process by invoking `task_timeslice()`. This function considers the static priority of the process and returns the corresponding base time quantum, according to the formula (1) shown in the earlier section "Scheduling of Conventional Processes." Moreover, the `first_time_slice` field of `current` is cleared: this flag is set by `copy_process()` in the service

routine of the fork() system call, and should be cleared as soon as the first time quantum of the process elapses.

Next, scheduler_tick() invokes the set_tsk_need_resched() function to set the TIF_NEED_RESCHED flag of the process. As described in the section "Returning from Interrupts and Exceptions" in Chapter 4, this flag forces the invocation of the schedule() function, so that current can be replaced by another real-time process having equal (or higher) priority, if any.

The last operation of scheduler_tick() consists of moving the process descriptor to the last position of the runqueue active list corresponding to the priority of current. Placing current in the last position ensures that it will not be selected again for execution until every real-time runnable process having the same priority as current will get a slice of the CPU time. This is the meaning of round-robin scheduling. The descriptor is moved by first invoking list_del() to remove the process from the runqueue active list, then by invoking list_add_tail() to insert back the process in the last position of the same list.

Updating the time slice of a conventional process

If the current process is a conventional process, the scheduler_tick() function performs the following operations:

1. Decreases the time slice counter (current->time_slice).
2. Checks the time slice counter. If the time quantum is exhausted, the function performs the following operations:
   a. Invokes dequeue_task() to remove current from the this_rq()->active set of runnable processes.
   b. Invokes set_tsk_need_resched() to set the TIF_NEED_RESCHED flag.
   c. Updates the dynamic priority of current:

      ```
      current->prio = effective_prio(current);
      ```

      The effective_prio() function reads the static_prio and sleep_avg fields of current, and computes the dynamic priority of the process according to the formula (2) shown in the earlier section "Scheduling of Conventional Processes."
   d. Refills the time quantum of the process:

      ```
      current->time_slice = task_timeslice(current);
      current->first_time_slice = 0;
      ```
   e. If the expired_timestamp field of the local runqueue data structure is equal to zero (that is, the set of expired processes is empty), writes into the field the value of the current tick:

      ```
      if (!this_rq()->expired_timestamp)
          this_rq()->expired_timestamp = jiffies;
      ```
   f. Inserts the current process either in the active set or in the expired set:

```
if (!TASK_INTERACTIVE(current) || EXPIRED_STARVING(this_rq()) {
    enqueue_task(current, this_rq()->expired);
    if (current->static_prio < this_rq()->best_expired_prio)
        this_rq()->best_expired_prio = current->static_prio;
} else
    enqueue_task(current, this_rq()->active);
```

The TASK_INTERACTIVE macro yields the value one if the process is recognized as interactive using the formula (3) shown in the earlier section "Scheduling of Conventional Processes." The EXPIRED_STARVING macro checks whether the first expired process in the runqueue had to wait for more than 1000 ticks times the number of runnable processes in the runqueue plus one; if so, the macro yields the value one. The EXPIRED_STARVING macro also yields the value one if the static priority value of the current process is greater than the static priority value of an already expired process.

3. Otherwise, if the time quantum is not exhausted (current->time_slice is not zero), checks whether the remaining time slice of the current process is too long:

```
if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
        p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
       (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
       (p->array == rq->active)) {
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
                  this_rq()->active->queue+current->prio);
    set_tsk_need_resched(p);
}
```

The TIMESLICE_GRANULARITY macro yields the product of the number of CPUs in the system and a constant proportional to the bonus of the current process (see Table 7-3 earlier in the chapter). Basically, the time quantum of interactive processes with high static priorities is split into several pieces of TIMESLICE_GRANULARITY size, so that they do not monopolize the CPU.

The try_to_wake_up( ) Function

The try_to_wake_up( ) function awakes a sleeping or stopped process by setting its state to TASK_RUNNING and inserting it into the runqueue of the local CPU. For instance, the function is invoked to wake up processes included in a wait queue (see the section "How Processes Are Organized" in Chapter 3) or to resume execution of processes waiting for a signal (see Chapter 11). The function receives as its parameters:

- The descriptor pointer (p) of the process to be awakened
- A mask of the process states (state) that can be awakened
- A flag (sync) that forbids the awakened process to preempt the process currently running on the local CPU

The function performs the following operations:

1. Invokes the `task_rq_lock()` function to disable local interrupts and to acquire the lock of the runqueue `rq` owned by the CPU that was last executing the process (it could be different from the local CPU). The logical number of that CPU is stored in the `p->thread_info->cpu` field.
2. Checks if the state of the process `p->state` belongs to the mask of states `state` passed as argument to the function; if this is not the case, it jumps to step 9 to terminate the function.
3. If the `p->array` field is not `NULL`, the process already belongs to a runqueue; therefore, it jumps to step 8.
4. In multiprocessor systems, it checks whether the process to be awakened should be migrated from the runqueue of the lastly executing CPU to the runqueue of another CPU. Essentially, the function selects a target runqueue according to some heuristic rules. For example:
   - If some CPU in the system is idle, it selects its runqueue as the target. Preference is given to the previously executing CPU and to the local CPU, in this order.
   - If the workload of the previously executing CPU is significantly lower than the workload of the local CPU, it selects the old runqueue as the target.
   - If the process has been executed recently, it selects the old runqueue as the target (the hardware cache might still be filled with the data of the process).
   - If moving the process to the local CPU reduces the unbalance between the CPUs, the target is the local runqueue (see the section "Runqueue Balancing in Multiprocessor Systems" later in this chapter).

   After this step has been executed, the function has identified a target CPU that will execute the awakened process and, correspondingly, a target runqueue `rq` in which to insert the process.
5. If the process is in the `TASK_UNINTERRUPTIBLE` state, it decreases the `nr_uninterruptible` field of the target runqueue, and sets the `p->activated` field of the process descriptor to -1. See the later section "The recalc_task_prio() Function" for an explanation of the `activated` field.
6. Invokes the `activate_task()` function, which in turn performs the following substeps:
   a. Invokes `sched_clock()` to get the current timestamp in nanoseconds. If the target CPU is not the local CPU, it compensates for the drift of the local timer interrupts by using the timestamps relative to the last occurrences of the timer interrupts on the local and target CPUs:

```
now = (sched_clock() - this_rq()->timestamp_last_tick)
      + rq->timestamp_last_tick;
```

b. Invokes `recalc_task_prio()`, passing to it the process descriptor pointer and the timestamp computed in the previous step. The `recalc_task_prio()` function is described in the next section.

c. Sets the value of the `p->activated` field according to Table 7-6 later in this chapter.

d. Sets the `p->timestamp` field with the timestamp computed in step 6a.

e. Inserts the process descriptor in the active set:

```
enqueue_task(p, rq->active);
rq->nr_running++;
```

7. If either the target CPU is not the local CPU or if the `sync` flag is not set, it checks whether the new runnable process has a dynamic priority higher than that of the current process of the `rq` runqueue (`p->prio < rq->curr->prio`); if so, invokes `resched_task()` to preempt `rq->curr`. In uniprocessor systems the latter function just executes `set_tsk_need_resched()` to set the `TIF_NEED_RESCHED` flag of the `rq->curr` process. In multiprocessor systems `resched_task()` also checks whether the old value of whether `TIF_NEED_RESCHED` flag was zero, the target CPU is different from the local CPU, and whether the `TIF_POLLING_NRFLAG` flag of the `rq->curr` process is clear (the target CPU is not actively polling the status of the `TIF_NEED_RESCHED` flag of the process). If so, `resched_task()` invokes `smp_send_reschedule()` to raise an IPI and force rescheduling on the target CPU (see the section "Interprocessor Interrupt Handling" in Chapter 4).
8. Sets the `p->state` field of the process to `TASK_RUNNING`.
9. Invokes `task_rq_unlock()` to unlock the `rq` runqueue and reenable the local interrupts.
10. Returns 1 (if the process has been successfully awakened) or 0 (if the process has not been awakened).

The recalc_task_prio() Function

The `recalc_task_prio()` function updates the average sleep time and the dynamic priority of a process. It receives as its parameters a process descriptor pointer `p` and a timestamp `now` computed by the `sched_clock()` function.

The function executes the following operations:

1. Stores in the `sleep_time` local variable the result of:

   $\min(\texttt{now} - \texttt{p->timestamp}, 10^9)$

   The `p->timestamp` field contains the timestamp of the process switch that put the process to sleep; therefore, `sleep_time` stores the number of nanoseconds that the process spent sleeping since its last execution (or the equivalent of 1 second, if the process slept more).

2. If sleep_time is not greater than zero, it jumps to step 8 so as to skip updating the average sleep time of the process.
3. Checks whether the process is not a kernel thread, whether it is awakening from the TASK_UNINTERRUPTIBLE state (p->activated field equal to −1; see step 5 in the previous section), and whether it has been continuously asleep beyond a given sleep time threshold. If these three conditions are fulfilled, the function sets the p->sleep_avg field to the equivalent of 900 ticks (an empirical value obtained by subtracting the duration of the base time quantum of a standard process from the maximum average sleep time). Then, it jumps to step 8.

   The sleep time threshold depends on the static priority of the process; some typical values are shown in Table 7-2. In short, the goal of this empirical rule is to ensure that processes that have been asleep for a long time in uninterruptible mode—usually waiting for disk I/O operations—get a predefined sleep average value that is large enough to allow them to be quickly serviced, but it is also not so large to cause starvation for other processes.
4. Executes the CURRENT_BONUS macro to compute the *bonus* value of the previous average sleep time of the process (see Table 7-3). If (10–*bonus*) is greater than zero, the function multiplies sleep_time by this value. Since sleep_time will be added to the average sleep time of the process (see step 6 below), the lower the current average sleep time is, the more rapidly it will rise.
5. If the process is in TASK_UNINTERRUPTIBLE mode and it is not a kernel thread, it performs the following substeps:
   a. Checks whether the average sleep time p->sleep_avg is greater than or equal to its sleep time threshold (see Table 7-2 earlier in this chapter). If so, it resets the sleep_time local variable to zero—thus skipping the adjustment of the average sleep time—and jumps to step 6.
   b. If the sum sleep_time+ p->sleep_avg is greater than or equal to the sleep time threshold, it sets the p->sleep_avg field to the sleep time threshold, and sets sleep_time to zero.

   By somewhat limiting the increment of the average sleep time of the process, the function does not reward too much batch processes that sleep for a long time.
6. Adds sleep_time to the average sleep time of the process (p->sleep_avg).
7. Checks whether p->sleep_avg exceeds 1000 ticks (in nanoseconds); if so, the function cuts it down to 1000 ticks (in nanoseconds).
8. Updates the dynamic priority of the process:

   ```
   p->prio = effective_prio(p);
   ```

   The effective_prio() function has already been discussed in the section "The scheduler_tick() Function" earlier in this chapter.

The schedule( ) Function

The schedule( ) function implements the scheduler. Its objective is to find a process in the runqueue list and then assign the CPU to it. It is invoked, directly or in a lazy (deferred) way, by several kernel routines.

Direct invocation

The scheduler is invoked directly when the current process must be blocked right away because the resource it needs is not available. In this case, the kernel routine that wants to block it proceeds as follows:

1. Inserts current in the proper wait queue.
2. Changes the state of current either to TASK_INTERRUPTIBLE or to TASK_UNINTERRUPTIBLE.
3. Invokes schedule( ).
4. Checks whether the resource is available; if not, goes to step 2.
5. Once the resource is available, removes current from the wait queue.

The kernel routine checks repeatedly whether the resource needed by the process is available; if not, it yields the CPU to some other process by invoking schedule( ). Later, when the scheduler once again grants the CPU to the process, the availability of the resource is rechecked. These steps are similar to those performed by wait_event( ) and similar functions described in the section "How Processes Are Organized" in Chapter 3.

The scheduler is also directly invoked by many device drivers that execute long iterative tasks. At each iteration cycle, the driver checks the value of the TIF_NEED_RESCHED flag and, if necessary, invokes schedule( ) to voluntarily relinquish the CPU.

Lazy invocation

The scheduler can also be invoked in a lazy way by setting the TIF_NEED_RESCHED flag of current to 1. Because a check on the value of this flag is always made before resuming the execution of a User Mode process (see the section "Returning from Interrupts and Exceptions" in Chapter 4), schedule( ) will definitely be invoked at some time in the near future.

Typical examples of lazy invocation of the scheduler are:

- When current has used up its quantum of CPU time; this is done by the scheduler_tick( ) function.
- When a process is woken up and its priority is higher than that of the current process; this task is performed by the try_to_wake_up( ) function.
- When a sched_setscheduler( ) system call is issued (see the section "System Calls Related to Scheduling" later in this chapter).

Actions performed by schedule( ) before a process switch

The goal of the schedule( ) function consists of replacing the currently executing process with another one. Thus, the key outcome of the function is to set a local variable called next, so that it points to the descriptor of the process selected to replace current. If no runnable process in the system has priority greater than the priority of current, at the end, next coincides with current and no process switch takes place.

The schedule( ) function starts by disabling kernel preemption and initializing a few local variables:

```
need_resched:

preempt_disable();
prev = current;
rq = this_rq();
```

As you see, the pointer returned by current is saved in prev, and the address of the runqueue data structure corresponding to the local CPU is saved in rq.

Next, schedule( ) makes sure that prev doesn't hold the big kernel lock (see the section "The Big Kernel Lock" in Chapter 5):

```
if (prev->lock_depth >= 0)
    up(&kernel_sem);
```

Notice that schedule( ) doesn't change the value of the lock_depth field; when prev resumes its execution, it reacquires the kernel_flag spin lock if the value of this field is not negative. Thus, the big kernel lock is automatically released and reacquired across a process switch.

The sched_clock( ) function is invoked to read the TSC and convert its value to nanoseconds; the timestamp obtained is saved in the now local variable. Then, schedule( ) computes the duration of the CPU time slice used by prev:

```
now = sched_clock();
run_time = now - prev->timestamp;
if (run_time > 1000000000)
    run_time = 1000000000;
```

The usual cut-off at 1 second (converted to nanoseconds) applies. The run_time value is used to charge the process for the CPU usage. However, a process having a high average sleep time is favored:

```
run_time /= (CURRENT_BONUS(prev) ? : 1);
```

Remember that CURRENT_BONUS returns a value between 0 and 10 that is proportional to the average sleep time of the process.

Before starting to look at the runnable processes, schedule( ) must disable the local interrupts and acquire the spin lock that protects the runqueue:

```
spin_lock_irq(&rq->lock);
```

As explained in the section "Process Termination" in Chapter 3, prev might be a process that is being terminated. To recognize this case, schedule( ) looks at the PF_DEAD flag:

```
if (prev->flags & PF_DEAD)
    prev->state = EXIT_DEAD;
```

Next, schedule( ) examines the state of prev. If it is not runnable and it has not been preempted in Kernel Mode (see the section "Returning from Interrupts and Exceptions" in Chapter 4), then it should be removed from the runqueue. However, if it has nonblocked pending signals and its state is TASK_INTERRUPTIBLE, the function sets the process state to TASK_RUNNING and leaves it into the runqueue. This action is not the same as assigning the processor to prev; it just gives prev a chance to be selected for execution:

```
if (prev->state != TASK_RUNNING &&
    !(preempt_count() & PREEMPT_ACTIVE)) {
    if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
        prev->state = TASK_RUNNING;
    else {
        if (prev->state == TASK_UNINTERRUPTIBLE)
            rq->nr_uninterruptible++;
        deactivate_task(prev, rq);
    }
}
```

The deactivate_task( ) function removes the process from the runqueue:

```
rq->nr_running--;
dequeue_task(p, p->array);
p->array = NULL;
```

Now, schedule( ) checks the number of runnable processes left in the runqueue. If there are some runnable processes, the function invokes the dependent_sleeper( ) function. In most cases, this function immediately returns zero. If, however, the kernel supports the hyper-threading technology (see the section "Runqueue Balancing in Multiprocessor Systems" later in this chapter), the function checks whether the process that is going to be selected for execution has significantly lower priority than a sibling process already running on a logical CPU of the same physical CPU; in this particular case, schedule( ) refuses to select the lower privilege process and executes the *swapper* process instead.

```
if (rq->nr_running) {
    if (dependent_sleeper(smp_processor_id(), rq)) {
        next = rq->idle;
        goto switch_tasks;
    }
}
```

If no runnable process exists, the function invokes idle_balance( ) to move some runnable process from another runqueue to the local runqueue; idle_balance( ) is simi-

lar to load_balance(), which is described in the later section "The load_balance() Function."

```
if (!rq->nr_running) {
    idle_balance(smp_processor_id(), rq);
    if (!rq->nr_running) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        wake_sleeping_dependent(smp_processor_id(), rq);
        if (!rq->nr_running)
            goto switch_tasks;
    }
}
```

If idle_balance() fails in moving some process in the local runqueue, schedule() invokes wake_sleeping_dependent() to reschedule runnable processes in *idle CPUs* (that is, in every CPU that runs the *swapper* process). As explained earlier when discussing the dependent_sleeper() function, this unusual case might happen when the kernel supports the hyper-threading technology. However, in uniprocessor systems, or when all attempts to move a runnable process in the local runqueue have failed, the function picks the *swapper* process as next and continues with the next phase.

Let's suppose that the schedule() function has determined that the runqueue includes some runnable processes; now it has to check that at least one of these runnable processes is active. If not, the function exchanges the contents of the active and expired fields of the runqueue data structure; thus, all expired processes become active, while the empty set is ready to receive the processes that will expire in the future.

```
array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = 140;
}
```

It is time to look up a runnable process in the active prio_array_t data structure (see the section "Identifying a Process" in Chapter 3). First of all, schedule( ) searches for the first nonzero bit in the bitmask of the active set. Remember that a bit in the bitmask is set when the corresponding priority list is not empty. Thus, the index of the first nonzero bit indicates the list containing the best process to run. Then, the first process descriptor in that list is retrieved:

```
idx = sched_find_first_bit(array->bitmap);
next = list_entry(array->queue[idx].next, task_t, run_list);
```

The sched_find_first_bit() function is based on the bsfl assembly language instruction, which returns the bit index of the least significant bit set to one in a 32-bit word.

The next local variable now stores the descriptor pointer of the process that will replace prev. The schedule() function looks at the next->activated field. This field encodes the state of the process when it was awakened, as illustrated in Table 7-6.

Table 7-6. The meaning of the activated field in the process descriptor

Value	Description
0	The process was in TASK_RUNNING state.
1	The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by a system call service routine or a kernel thread.
2	The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by an interrupt handler or a deferrable function.
–1	The process was in TASK_UNINTERRUPTIBLE state and it is being awakened.

If next is a conventional process and it is being awakened from the TASK_INTERRUPTIBLE or TASK_STOPPED state, the scheduler adds to the average sleep time of the process the nanoseconds elapsed since the process was inserted into the runqueue. In other words, the sleep time of the process is increased to cover also the time spent by the process in the runqueue waiting for the CPU:

```
if (next->prio >= 100 && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;
    if (next->activated == 1)
        delta = (delta * 38) / 128;
    array = next->array;
    dequeue_task(next, array);
    recalc_task_prio(next, next->timestamp + delta);
    enqueue_task(next, array);
}
next->activated = 0;
```

Observe that the scheduler makes a distinction between a process awakened by an interrupt handler or deferrable function, and a process awakened by a system call service routine or a kernel thread. In the former case, the scheduler adds the whole runqueue waiting time, while in the latter it adds just a fraction of that time. This is because interactive processes are more likely to be awakened by asynchronous events (think of the user pressing keys on the keyboard) rather than by synchronous ones.

Actions performed by schedule() to make the process switch

Now the schedule() function has determined the next process to run. In a moment, the kernel will access the thread_info data structure of next, whose address is stored close to the top of next's process descriptor:

```
switch_tasks:

prefetch(next);
```

The prefetch macro is a hint to the CPU control unit to bring the contents of the first fields of next's process descriptor in the hardware cache. It is here just to improve the

performance of schedule(), because the data are moved in parallel to the execution of the following instructions, which do not affect next.

Before replacing prev, the scheduler should do some administrative work:

```
clear_tsk_need_resched(prev);
rcu_qsctr_inc(prev->thread_info->cpu);
```

The clear_tsk_need_resched() function clears the TIF_NEED_RESCHED flag of prev, just in case schedule() has been invoked in the lazy way. Then, the function records that the CPU is going through a quiescent state (see the section "Read-Copy Update (RCU)" in Chapter 5).

The schedule() function must also decrease the average sleep time of prev, charging to it the slice of CPU time used by the process:

```
prev->sleep_avg -= run_time;
if ((long)prev->sleep_avg <= 0)
    prev->sleep_avg = 0;
prev->timestamp = prev->last_ran = now;
```

The timestamps of the process are then updated.

It is quite possible that prev and next are the same process: this happens if no other higher or equal priority active process is present in the runqueue. In this case, the function skips the process switch:

```
if (prev == next) {
    spin_unlock_irq(&rq->lock);
    goto finish_schedule;
}
```

At this point, prev and next are different processes, and the process switch is for real:

```
next->timestamp = now;
rq->nr_switches++;
rq->curr = next;
prev = context_switch(rq, prev, next);
```

The context_switch() function sets up the address space of next. As we'll see in "Memory Descriptor of Kernel Threads" in Chapter 9, the active_mm field of the process descriptor points to the memory descriptor that is used by the process, while the mm field points to the memory descriptor owned by the process. For normal processes, the two fields hold the same address; however, a kernel thread does not have its own address space and its mm field is always set to NULL. The context_switch() function ensures that if next is a kernel thread, it uses the address space used by prev:

```
if (!next->mm) {
    next->active_mm = prev->active_mm;
    atomic_inc(&prev->active_mm->mm_count);
    enter_lazy_tlb(prev->active_mm, next);
}
```

Up to Linux version 2.2, kernel threads had their own address space. That design choice was suboptimal, because the Page Tables had to be changed whenever the

scheduler selected a new process, even if it was a kernel thread. Because kernel threads run in Kernel Mode, they use only the fourth gigabyte of the linear address space, whose mapping is the same for all processes in the system. Even worse, writing into the cr3 register invalidates all TLB entries (see "Translation Lookaside Buffers (TLB)" in Chapter 2), which leads to a significant performance penalty. Linux is nowadays much more efficient because Page Tables aren't touched at all if next is a kernel thread. As further optimization, if next is a kernel thread, the schedule() function sets the process into lazy TLB mode (see the section "Translation Lookaside Buffers (TLB)" in Chapter 2).

Conversely, if next is a regular process, the context_switch() function replaces the address space of prev with the one of next:

```
if (next->mm)
    switch_mm(prev->active_mm, next->mm, next);
```

If prev is a kernel thread or an exiting process, the context_switch() function saves the pointer to the memory descriptor used by prev in the runqueue's prev_mm field, then resets prev->active_mm:

```
if (!prev->mm) {
    rq->prev_mm = prev->active_mm;
    prev->active_mm = NULL;
}
```

Now context_switch() can finally invoke switch_to() to perform the process switch between prev and next (see the section "Performing the Process Switch" in Chapter 3):

```
switch_to(prev, next, prev);
return prev;
```

Actions performed by schedule() after a process switch

The instructions of the context_switch() and schedule() functions following the switch_to macro invocation will not be performed right away by the next process, but at a later time by prev when the scheduler selects it again for execution. However, at that moment, the prev local variable does not point to our original process that was to be replaced when we started the description of schedule(), but rather to the process that was replaced by our original prev when it was scheduled again. (If you are confused, go back and read the section "Performing the Process Switch" in Chapter 3.) The first instructions after a process switch are:

```
barrier();
finish_task_switch(prev);
```

Right after the invocation of the context_switch() function in schedule(), the barrier() macro yields an optimization barrier for the code (see the section "Optimization and Memory Barriers" in Chapter 5). Then, the finish_task_switch() function is executed:

```
mm = this_rq()->prev_mm;
this_rq()->prev_mm = NULL;
```

```
prev_task_flags = prev->flags;
spin_unlock_irq(&this_rq()->lock);
if (mm)
    mmdrop(mm);
if (prev_task_flags & PF_DEAD)
    put_task_struct(prev);
```

If prev is a kernel thread, the prev_mm field of the runqueue stores the address of the memory descriptor that was lent to prev. As we'll see in Chapter 9, mmdrop() decreases the usage counter of the memory descriptor; if the counter reaches 0 (likely because prev is a zombie process), the function also frees the descriptor together with the associated Page Tables and virtual memory regions.

The finish_task_switch() function also releases the spin lock of the runqueue and enables the local interrupts. Then, it checks whether prev is a zombie task that is being removed from the system (see the section "Process Termination" in Chapter 3); if so, it invokes put_task_struct() to free the process descriptor reference counter and drop all remaining references to the process (see the section "Process Removal" in Chapter 3).

The very last instructions of the schedule() function are:

```
finish_schedule:

prev = current;
if (prev->lock_depth >= 0)
    __reacquire_kernel_lock();
preempt_enable_no_resched();
if (test_bit(TIF_NEED_RESCHED, &current_thread_info()->flags)
   goto need_resched;
return;
```

As you see, schedule() reacquires the big kernel lock if necessary, reenables kernel preemption, and checks whether some other process has set the TIF_NEED_RESCHED flag of the current process. In this case, the whole schedule() function is reexecuted from the beginning; otherwise, the function terminates.

Runqueue Balancing in Multiprocessor Systems

We have seen in Chapter 4 that Linux sticks to the Symmetric Multiprocessing model (SMP); this means, essentially, that the kernel should not have any bias toward one CPU with respect to the others. However, multiprocessor machines come in many different flavors, and the scheduler behaves differently depending on the hardware characteristics. In particular, we will consider the following three types of multiprocessor machines:

Classic multiprocessor architecture
: Until recently, this was the most common architecture for multiprocessor machines. These machines have a common set of RAM chips shared by all CPUs.

Hyper-threading

A hyper-threaded chip is a microprocessor that executes several threads of execution at once; it includes several copies of the internal registers and quickly switches between them. This technology, which was invented by Intel, allows the processor to exploit the machine cycles to execute another thread while the current thread is stalled for a memory access. A hyper-threaded physical CPU is seen by Linux as several different logical CPUs.

NUMA

CPUs and RAM chips are grouped in local "nodes" (usually a node includes one CPU and a few RAM chips). The memory arbiter (a special circuit that serializes the accesses to RAM performed by the CPUs in the system, see the section "Memory Addresses" in Chapter 2) is a bottleneck for the performance of the classic multiprocessor systems. In a NUMA architecture, when a CPU accesses a "local" RAM chip inside its own node, there is little or no contention, thus the access is usually fast; on the other hand, accessing a "remote" RAM chip outside of its node is much slower. We'll mention in the section "Non-Uniform Memory Access (NUMA)" in Chapter 8 how the Linux kernel memory allocator supports NUMA architectures.

These basic kinds of multiprocessor systems are often combined. For instance, a motherboard that includes two different hyper-threaded CPUs is seen by the kernel as four logical CPUs.

As we have seen in the previous section, the `schedule()` function picks the new process to run from the runqueue of the local CPU. Therefore, a given CPU can execute only the runnable processes that are contained in the corresponding runqueue. On the other hand, a runnable process is always stored in exactly one runqueue: no runnable process ever appears in two or more runqueues. Therefore, until a process remains runnable, it is usually bound to one CPU.

This design choice is usually beneficial for system performance, because the hardware cache of every CPU is likely to be filled with data owned by the runnable processes in the runqueue. In some cases, however, binding a runnable process to a given CPU might induce a severe performance penalty. For instance, consider a large number of batch processes that make heavy use of the CPU: if most of them end up in the same runqueue, one CPU in the system will be overloaded, while the others will be nearly idle.

Therefore, the kernel periodically checks whether the workloads of the runqueues are balanced and, if necessary, moves some process from one runqueue to another. However, to get the best performance from a multiprocessor system, the load balancing algorithm should take into consideration the topology of the CPUs in the system. Starting from kernel version 2.6.7, Linux sports a sophisticated runqueue balancing algorithm based on the notion of "scheduling domains." Thanks to the scheduling domains, the algorithm can be easily tuned for all kinds of existing multiprocessor

architectures (and even for recent architectures such as those based on the "multi-core" microprocessors).

Scheduling Domains

Essentially, a *scheduling domain* is a set of CPUs whose workloads should be kept balanced by the kernel. Generally speaking, scheduling domains are hierarchically organized: the top-most scheduling domain, which usually spans all CPUs in the system, includes children scheduling domains, each of which include a subset of the CPUs. Thanks to the hierarchy of scheduling domains, workload balancing can be done in a rather efficient way.

Every scheduling domain is partitioned, in turn, in one or more *groups*, each of which represents a subset of the CPUs of the scheduling domain. Workload balancing is always done between groups of a scheduling domain. In other words, a process is moved from one CPU to another only if the total workload of some group in some scheduling domain is significantly lower than the workload of another group in the same scheduling domain.

Figure 7-2 illustrates three examples of scheduling domain hierarchies, corresponding to the three main architectures of multiprocessor machines.

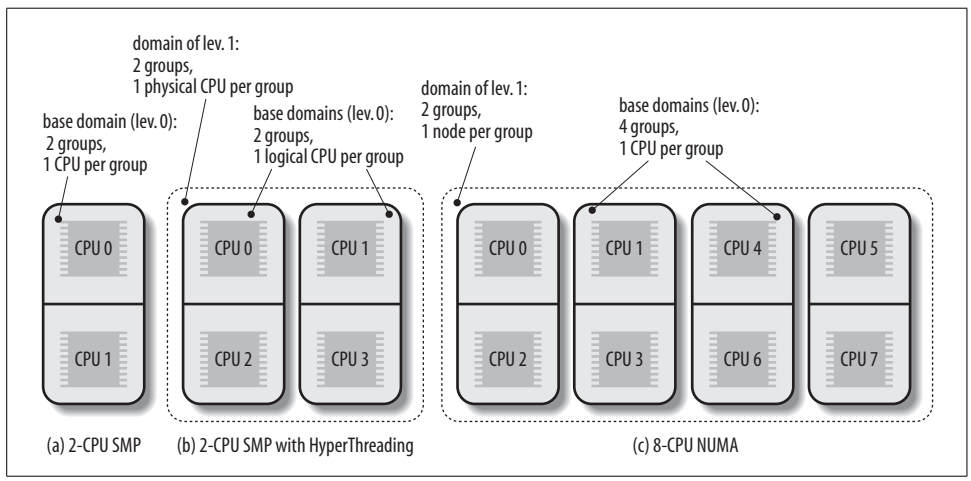


Figure 7-2. Three examples of scheduling domain hierarchies

Figure 7-2 (a) represents a hierarchy composed by a single scheduling domain for a 2-CPU classic multiprocessor architecture. The scheduling domain includes only two groups, each of which includes one CPU.

Figure 7-2 (b) represents a two-level hierarchy for a 2-CPU multiprocessor box with hyper-threading technology. The top-level scheduling domain spans all four logical CPUs in the system, and it is composed by two groups. Each group of the top-level

domain corresponds to a child scheduling domain and spans a physical CPU. The bottom-level scheduling domains (also called *base scheduling domains*) include two groups, one for each logical CPU.

Finally, Figure 7-2 (c) represents a two-level hierarchy for an 8-CPU NUMA architecture with two nodes and four CPUs per node. The top-level domain is organized in two groups, each of which corresponds to a different node. Every base scheduling domain spans the CPUs inside a single node and has four groups, each of which spans a single CPU.

Every scheduling domain is represented by a `sched_domain` descriptor, while every group inside a scheduling domain is represented by a `sched_group` descriptor. Each `sched_domain` descriptor includes a field `groups`, which points to the first element in a list of group descriptors. Moreover, the `parent` field of the `sched_domain` structure points to the descriptor of the parent scheduling domain, if any.

The `sched_domain` descriptors of all physical CPUs in the system are stored in the per-CPU variable `phys_domains`. If the kernel does not support the hyper-threading technology, these domains are at the bottom level of the domain hierarchy, and the `sd` fields of the runqueue descriptors point to them—that is, they are the base scheduling domains. Conversely, if the kernel supports the hyper-threading technology, the bottom-level scheduling domains are stored in the per-CPU variable `cpu_domains`.

The rebalance_tick() Function

To keep the runqueues in the system balanced, the `rebalance_tick()` function is invoked by `scheduler_tick()` once every tick. It receives as its parameters the index `this_cpu` of the local CPU, the address `this_rq` of the local runqueue, and a flag, `idle`, which can assume the following values:

`SCHED_IDLE`
: The CPU is currently idle, that is, `current` is the *swapper* process.

`NOT_IDLE`
: The CPU is not currently idle, that is, `current` is not the *swapper* process.

The `rebalance_tick()` function determines first the number of processes in the runqueue and updates the runqueue's average workload; to do this, the function accesses the `nr_running` and `cpu_load` fields of the runqueue descriptor.

Then, `rebalance_tick()` starts a loop over all scheduling domains in the path from the base domain (referenced by the `sd` field of the local runqueue descriptor) to the top-level domain. In each iteration the function determines whether the time has come to invoke the `load_balance()` function, thus executing a rebalancing operation on the scheduling domain. The value of `idle` and some parameters stored in the `sched_domain` descriptor determine the frequency of the invocations of `load_balance()`. If `idle` is equal to `SCHED_IDLE`, then the runqueue is empty, and `rebalance_tick()`

invokes load_balance() quite often (roughly once every one or two ticks for scheduling domains corresponding to logical and physical CPUs). Conversely, if idle is equal to NOT_IDLE, rebalance_tick() invokes load_balance() sparingly (roughly once every 10 milliseconds for scheduling domains corresponding to logical CPUs, and once every 100 milliseconds for scheduling domains corresponding to physical CPUs).

The load_balance() Function

The load_balance() function checks whether a scheduling domain is significantly unbalanced; more precisely, it checks whether unbalancing can be reduced by moving some processes from the busiest group to the runqueue of the local CPU. If so, the function attempts this migration. It receives four parameters:

this_cpu
: The index of the local CPU

this_rq
: The address of the descriptor of the local runqueue

sd
: Points to the descriptor of the scheduling domain to be checked

idle
: Either SCHED_IDLE (local CPU is idle) or NOT_IDLE

The function performs the following operations:

1. Acquires the this_rq->lock spin lock.
2. Invokes the find_busiest_group() function to analyze the workloads of the groups inside the scheduling domain. The function returns the address of the sched_group descriptor of the busiest group, provided that this group does not include the local CPU; in this case, the function also returns the number of processes to be moved into the local runqueue to restore balancing. On the other hand, if either the busiest group includes the local CPU or all groups are essentially balanced, the function returns NULL. This procedure is not trivial, because the function tries to filter the statistical fluctuations in the workloads.
3. If find_busiest_group() did not find a group not including the local CPU that is significantly busier than the other groups in the scheduling domain, the function releases the this_rq->lock spin lock, tunes the parameters in the scheduling domain descriptor so as to delay the next invocation of load_balance() on the local CPU, and terminates.
4. Invokes the find_busiest_queue() function to find the busiest CPUs in the group found in step 2. The function returns the descriptor address busiest of the corresponding runqueue.
5. Acquires a second spin lock, namely the busiest->lock spin lock. To prevent deadlocks, this has to be done carefully: the this_rq->lock is first released, then the two locks are acquired by increasing CPU indices.

6. Invokes the move_tasks() function to try moving some processes from the busiest runqueue to the local runqueue this_rq (see the next section).
7. If the move_task() function failed in migrating some process to the local runqueue, the scheduling domain is still unbalanced. Sets to 1 the busiest->active_balance flag and wakes up the *migration* kernel thread whose descriptor is stored in busiest->migration_thread. The *migration* kernel thread walks the chain of the scheduling domain, from the base domain of the busiest runqueue to the top domain, looking for an idle CPU. If an idle CPU is found, the kernel thread invokes move_tasks() to move one process into the idle runqueue.
8. Releases the busiest->lock and this_rq->lock spin locks.
9. Terminates.

The move_tasks() Function

The move_tasks() function moves processes from a source runqueue to the local runqueue. It receives six parameters: this_rq and this_cpu (the local runqueue descriptor and the local CPU index), busiest (the source runqueue descriptor), max_nr_move (the maximum number of processes to be moved), sd (the address of the scheduling domain descriptor in which this balancing operation is carried on), and the idle flag (beside SCHED_IDLE and NOT_IDLE, this flag can also be set to NEWLY_IDLE when the function is indirectly invoked by idle_balance(); see the section "The schedule() Function" earlier in this chapter).

The function first analyzes the expired processes of the busiest runqueue, starting from the higher priority ones. When all expired processes have been scanned, the function scans the active processes of the busiest runqueue. For each candidate process, the function invokes can_migrate_task(), which returns 1 if all the following conditions hold:

- The process is not being currently executed by the remote CPU.
- The local CPU is included in the cpus_allowed bitmask of the process descriptor.
- At least one of the following holds:
  - The local CPU is idle. If the kernel supports the hyper-threading technology, all logical CPUs in the local physical chip must be idle.
  - The kernel is having trouble in balancing the scheduling domain, because repeated attempts to move processes have failed.
  - The process to be moved is not "cache hot" (it has not recently executed on the remote CPU, so one can assume that no data of the process is included in the hardware cache of the remote CPU).

If can_migrate_task() returns the value 1, move_tasks() invokes the pull_task() function to move the candidate process to the local runqueue. Essentially, pull_task() executes dequeue_task() to remove the process from the remote runqueue, then executes enqueue_task() to insert the process in the local runqueue, and finally,

if the process just moved has higher dynamic priority than current, invokes resched_task() to preempt the current process of the local CPU.

System Calls Related to Scheduling

Several system calls have been introduced to allow processes to change their priorities and scheduling policies. As a general rule, users are always allowed to lower the priorities of their processes. However, if they want to modify the priorities of processes belonging to some other user or if they want to increase the priorities of their own processes, they must have superuser privileges.

The nice( ) System Call

The nice()* system call allows processes to change their base priority. The integer value contained in the increment parameter is used to modify the nice field of the process descriptor. The *nice* Unix command, which allows users to run programs with modified scheduling priority, is based on this system call.

The sys_nice() service routine handles the nice() system call. Although the increment parameter may have any value, absolute values larger than 40 are trimmed down to 40. Traditionally, negative values correspond to requests for priority increments and require superuser privileges, while positive ones correspond to requests for priority decreases. In the case of a negative increment, the function invokes the capable() function to verify whether the process has a CAP_SYS_NICE capability. Moreover, the function invokes the security_task_setnice() security hook. We discuss that function in Chapter 20. If the user turns out to have the privilege required to change priorities, sys_nice() converts current->static_prio to the range of nice values, adds the value of increment, and invokes the set_user_nice() function. In turn, the latter function gets the local runqueue lock, updates the static priority of current, invokes the resched_task() function to allow other processes to preempt current, and release the runqueue lock.

The nice() system call is maintained for backward compatibility only; it has been replaced by the setpriority() system call described next.

The getpriority( ) and setpriority( ) System Calls

The nice() system call affects only the process that invokes it. Two other system calls, denoted as getpriority() and setpriority(), act on the base priorities of all processes in a given group. getpriority() returns 20 minus the lowest nice field value among all processes in a given group—that is, the highest priority among those

* Because this system call is usually invoked to lower the priority of a process, users who invoke it for their processes are "nice" to other users.

processes; setpriority( ) sets the base priority of all processes in a given group to a given value.

The kernel implements these system calls by means of the sys_getpriority( ) and sys_setpriority( ) service routines. Both of them act essentially on the same group of parameters:

which
: The value that identifies the group of processes; it can assume one of the following:

  PRIO_PROCESS
  : Selects the processes according to their process ID (pid field of the process descriptor).

  PRIO_PGRP
  : Selects the processes according to their group ID (pgrp field of the process descriptor).

  PRIO_USER
  : Selects the processes according to their user ID (uid field of the process descriptor).

who
: The value of the pid, pgrp, or uid field (depending on the value of which) to be used for selecting the processes. If who is 0, its value is set to that of the corresponding field of the current process.

niceval
: The new base priority value (needed only by sys_setpriority( )). It should range between –20 (highest priority) and +19 (lowest priority).

As stated before, only processes with a CAP_SYS_NICE capability are allowed to increase their own base priority or to modify that of other processes.

As we will see in Chapter 10, system calls return a negative value only if some error occurred. For this reason, getpriority( ) does not return a normal nice value ranging between –20 and +19, but rather a nonnegative value ranging between 1 and 40.

The sched_getaffinity( ) and sched_setaffinity( ) System Calls

The sched_getaffinity( ) and sched_setaffinity( ) system calls respectively return and set up the CPU affinity mask of a process—the bit mask of the CPUs that are allowed to execute the process. This mask is stored in the cpus_allowed field of the process descriptor.

The sys_sched_getaffinity( ) system call service routine looks up the process descriptor by invoking find_task_by_pid( ), and then returns the value of the corresponding cpus_allowed field ANDed with the bitmap of the available CPUs.

The sys_sched_setaffinity( ) system call is a bit more complicated. Besides looking for the descriptor of the target process and updating the cpus_allowed field, this function has to check whether the process is included in a runqueue of a CPU that is no longer present in the new affinity mask. In the worst case, the process has to be moved from one runqueue to another one. To avoid problems due to deadlocks and race conditions, this job is done by the *migration* kernel threads (there is one thread per CPU). Whenever a process has to be moved from a runqueue rq1 to another runqueue rq2, the system call awakes the migration thread of rq1 (rq1->migration_thread), which in turn removes the process from rq1 and inserts it into rq2.

System Calls Related to Real-Time Processes

We now introduce a group of system calls that allow processes to change their scheduling discipline and, in particular, to become real-time processes. As usual, a process must have a CAP_SYS_NICE capability to modify the values of the rt_priority and policy process descriptor fields of any process, including itself.

The sched_getscheduler( ) and sched_setscheduler( ) system calls

The sched_getscheduler( ) system call queries the scheduling policy currently applied to the process identified by the pid parameter. If pid equals 0, the policy of the calling process is retrieved. On success, the system call returns the policy for the process: SCHED_FIFO, SCHED_RR, or SCHED_NORMAL (the latter is also called SCHED_OTHER). The corresponding sys_sched_getscheduler( ) service routine invokes find_process_by_pid( ), which locates the process descriptor corresponding to the given pid and returns the value of its policy field.

The sched_setscheduler( ) system call sets both the scheduling policy and the associated parameters for the process identified by the parameter pid. If pid is equal to 0, the scheduler parameters of the calling process will be set.

The corresponding sys_sched_setscheduler( ) system call service routine simply invokes do_sched_setscheduler( ). The latter function checks whether the scheduling policy specified by the policy parameter and the new priority specified by the param->sched_priority parameter are valid. It also checks whether the process has CAP_SYS_NICE capability or whether its owner has superuser rights. If everything is OK, it removes the process from its runqueue (if it is runnable); updates the static, real-time, and dynamic priorities of the process; inserts the process back in the runqueue; and finally invokes, if necessary, the resched_task( ) function to preempt the current process of the runqueue.

The sched_getparam( ) and sched_setparam( ) system calls

The sched_getparam( ) system call retrieves the scheduling parameters for the process identified by pid. If pid is 0, the parameters of the current process are retrieved.

The corresponding sys_sched_getparam( ) service routine, as one would expect, finds the process descriptor pointer associated with pid, stores its rt_priority field in a local variable of type sched_param, and invokes copy_to_user( ) to copy it into the process address space at the address specified by the param parameter.

The sched_setparam( ) system call is similar to sched_setscheduler( ). The difference is that sched_setparam( ) does not let the caller set the policy field's value.* The corresponding sys_sched_setparam( ) service routine invokes do_sched_setscheduler( ), with almost the same parameters as sys_sched_setscheduler( ).

The sched_yield( ) system call

The sched_yield( ) system call allows a process to relinquish the CPU voluntarily without being suspended; the process remains in a TASK_RUNNING state, but the scheduler puts it either in the expired set of the runqueue (if the process is a conventional one), or at the end of the runqueue list (if the process is a real-time one). The schedule( ) function is then invoked. In this way, other processes that have the same dynamic priority have a chance to run. The call is used mainly by SCHED_FIFO real-time processes.

The sched_get_priority_min( ) and sched_get_priority_max( ) system calls

The sched_get_priority_min( ) and sched_get_priority_max( ) system calls return, respectively, the minimum and the maximum real-time static priority value that can be used with the scheduling policy identified by the policy parameter.

The sys_sched_get_priority_min( ) service routine returns 1 if current is a real-time process, 0 otherwise.

The sys_sched_get_priority_max( ) service routine returns 99 (the highest priority) if current is a real-time process, 0 otherwise.

The sched_rr_get_interval( ) system call

The sched_rr_get_interval( ) system call writes into a structure stored in the User Mode address space the Round Robin time quantum for the real-time process identified by the pid parameter. If pid is zero, the system call writes the time quantum of the current process.

The corresponding sys_sched_rr_get_interval( ) service routine invokes, as usual, find_process_by_pid( ) to retrieve the process descriptor associated with pid. It then converts the base time quantum of the selected process into seconds and nanoseconds and copies the numbers into the User Mode structure. Conventionally, the time quantum of a FIFO real-time process is equal to zero.

* This anomaly is caused by a specific requirement of the POSIX standard.

CHAPTER 8

Memory Management

We saw in Chapter 2 how Linux takes advantage of 80×86's segmentation and paging circuits to translate logical addresses into physical ones. We also mentioned that some portion of RAM is permanently assigned to the kernel and used to store both the kernel code and the static kernel data structures.

The remaining part of the RAM is called *dynamic memory*. It is a valuable resource, needed not only by the processes but also by the kernel itself. In fact, the performance of the entire system depends on how efficiently dynamic memory is managed. Therefore, all current multitasking operating systems try to optimize the use of dynamic memory, assigning it only when it is needed and freeing it as soon as possible. Figure 8-1 shows schematically the page frames used as dynamic memory; see the section "Physical Memory Layout" in Chapter 2 for details.

This chapter, which consists of three main sections, describes how the kernel allocates dynamic memory for its own use. The sections "Page Frame Management" and "Memory Area Management" illustrate two different techniques for handling physically contiguous memory areas, while the section "Noncontiguous Memory Area Management" illustrates a third technique that handles noncontiguous memory areas. In these sections we'll cover topics such as memory zones, kernel mappings, the buddy system, the slab cache, and memory pools.

Page Frame Management

We saw in the section "Paging in Hardware" in Chapter 2 how the Intel Pentium processor can use two different page frame sizes: 4 KB and 4 MB (or 2 MB if PAE is enabled—see the section "The Physical Address Extension (PAE) Paging Mechanism" in Chapter 2). Linux adopts the smaller 4 KB page frame size as the standard memory allocation unit. This makes things simpler for two reasons:

- The Page Fault exceptions issued by the paging circuitry are easily interpreted. Either the page requested exists but the process is not allowed to address it, or

the page does not exist. In the second case, the memory allocator must find a free 4 KB page frame and assign it to the process.

- Although both 4 KB and 4 MB are multiples of all disk block sizes, transfers of data between main memory and disks are in most cases more efficient when the smaller size is used.

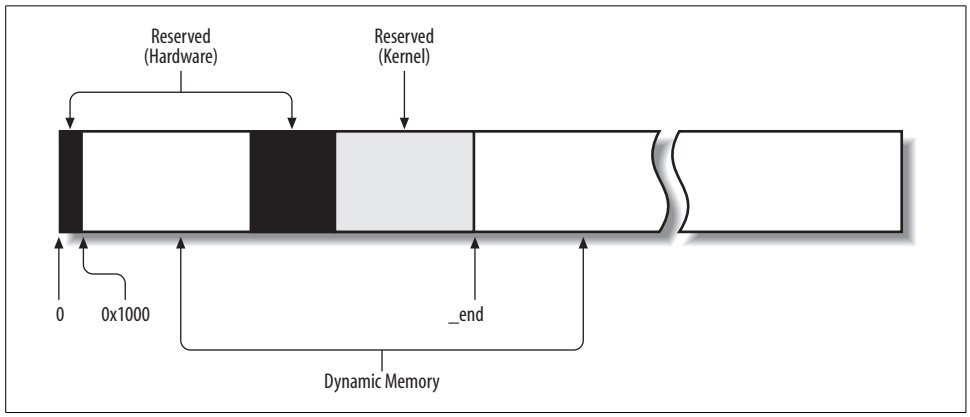


Figure 8-1. Dynamic memory

Page Descriptors

The kernel must keep track of the current status of each page frame. For instance, it must be able to distinguish the page frames that are used to contain pages that belong to processes from those that contain kernel code or kernel data structures. Similarly, it must be able to determine whether a page frame in dynamic memory is free. A page frame in dynamic memory is free if it does not contain any useful data. It is not free when the page frame contains data of a User Mode process, data of a software cache, dynamically allocated kernel data structures, buffered data of a device driver, code of a kernel module, and so on.

State information of a page frame is kept in a page descriptor of type `page`, whose fields are shown in Table 8-1. All page descriptors are stored in the `mem_map` array. Because each descriptor is 32 bytes long, the space required by `mem_map` is slightly less than 1% of the whole RAM. The `virt_to_page(addr)` macro yields the address of the page descriptor associated with the linear address `addr`. The `pfn_to_page(pfn)` macro yields the address of the page descriptor associated with the page frame having number `pfn`.

Table 8-1. The fields of the page descriptor

Type	Name	Description
`unsigned long`	`flags`	Array of flags (see Table 8-2). Also encodes the zone number to which the page frame belongs.
`atomic_t`	`_count`	Page frame's reference counter.

Table 8-1. The fields of the page descriptor (continued)

Type	Name	Description
atomic_t	_mapcount	Number of Page Table entries that refer to the page frame (-1 if none).
unsigned long	private	Available to the kernel component that is using the page (for instance, it is a buffer head pointer in case of buffer page; see "Block Buffers and Buffer Heads" in Chapter 15). If the page is free, this field is used by the buddy system (see later in this chapter).
struct address_space *	mapping	Used when the page is inserted into the page cache (see the section "The Page Cache" in Chapter 15), or when it belongs to an anonymous region (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
unsigned long	index	Used by several kernel components with different meanings. For instance, it identifies the position of the data stored in the page frame within the page's disk image or within an anonymous region (Chapter 15), or it stores a swapped-out page identifier (Chapter 17).
struct list_head	lru	Contains pointers to the least recently used doubly linked list of pages.

You don't have to fully understand the role of all fields in the page descriptor right now. In the following chapters, we often come back to the fields of the page descriptor. Moreover, several fields have different meaning, according to whether the page frame is free or what kernel component is using the page frame.

Let's describe in greater detail two of the fields:

`_count`
: A usage reference counter for the page. If it is set to -1, the corresponding page frame is free and can be assigned to any process or to the kernel itself. If it is set to a value greater than or equal to 0, the page frame is assigned to one or more processes or is used to store some kernel data structures. The `page_count()` function returns the value of the `_count` field increased by one, that is, the number of users of the page.

`flags`
: Includes up to 32 flags (see Table 8-2) that describe the status of the page frame. For each `PG_`*xyz* flag, the kernel defines some macros that manipulate its value. Usually, the `Page`*Xyz* macro returns the value of the flag, while the `SetPage`*Xyz* and `ClearPage`*Xyz* macro set and clear the corresponding bit, respectively.

Table 8-2. Flags describing the status of a page frame

Flag name	Meaning
PG_locked	The page is locked; for instance, it is involved in a disk I/O operation.
PG_error	An I/O error occurred while transferring the page.
PG_referenced	The page has been recently accessed.
PG_uptodate	This flag is set after completing a read operation, unless a disk I/O error happened.
PG_dirty	The page has been modified (see the section "Implementing the PFRA" in Chapter 17).

Table 8-2. Flags describing the status of a page frame (continued)

Flag name	Meaning
PG_lru	The page is in the active or inactive page list (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).
PG_active	The page is in the active page list (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).
PG_slab	The page frame is included in a slab (see the section "Memory Area Management" later in this chapter).
PG_highmem	The page frame belongs to the ZONE_HIGHMEM zone (see the following section "Non-Uniform Memory Access (NUMA)").
PG_checked	Used by some filesystems such as Ext2 and Ext3 (see Chapter 18).
PG_arch_1	Not used on the 80×86 architecture.
PG_reserved	The page frame is reserved for kernel code or is unusable.
PG_private	The private field of the page descriptor stores meaningful data.
PG_writeback	The page is being written to disk by means of the writepage method (see Chapter 16) .
PG_nosave	Used for system suspend/resume.
PG_compound	The page frame is handled through the extended paging mechanism (see the section "Extended Paging" in Chapter 2).
PG_swapcache	The page belongs to the swap cache (see the section "The Swap Cache" in Chapter 17).
PG_mappedtodisk	All data in the page frame corresponds to blocks allocated on disk.
PG_reclaim	The page has been marked to be written to disk in order to reclaim memory.
PG_nosave_free	Used for system suspend/resume.

Non-Uniform Memory Access (NUMA)

We are used to thinking of the computer's memory as a homogeneous, shared resource. Disregarding the role of the hardware caches, we expect the time required for a CPU to access a memory location to be essentially the same, regardless of the location's physical address and the CPU. Unfortunately, this assumption is not true in some architectures. For instance, it is not true for some multiprocessor Alpha or MIPS computers.

Linux 2.6 supports the *Non-Uniform Memory Access* (*NUMA*) model, in which the access times for different memory locations from a given CPU may vary. The physical memory of the system is partitioned in several *nodes*. The time needed by a given CPU to access pages within a single node is the same. However, this time might not be the same for two different CPUs. For every CPU, the kernel tries to minimize the number of accesses to costly nodes by carefully selecting where the kernel data structures that are most often referenced by the CPU are stored.*

* Furthermore, the Linux kernel makes use of NUMA even for some peculiar uniprocessor systems that have huge "holes" in the physical address space. The kernel handles these architectures by assigning the contiguous subranges of valid physical addresses to different memory nodes.

The physical memory inside each node can be split into several zones, as we will see in the next section. Each node has a descriptor of type pg_data_t, whose fields are shown in Table 8-3. All node descriptors are stored in a singly linked list, whose first element is pointed to by the pgdat_list variable.

Table 8-3. The fields of the node descriptor

Type	Name	Description
struct zone[]	node_zones	Array of zone descriptors of the node
struct zonelist[]	node_zonelists	Array of zonelist data structures used by the page allocator (see the later section "Memory Zones")
int	nr_zones	Number of zones in the node
struct page *	node_mem_map	Array of page descriptors of the node
struct bootmem_data *	bdata	Used in the kernel initialization phase
unsigned long	node_start_pfn	Index of the first page frame in the node
unsigned long	node_present_pages	Size of the memory node, excluding holes (in page frames)
unsigned long	node_spanned_pages	Size of the node, including holes (in page frames)
int	node_id	Identifier of the node
pg_data_t *	pgdat_next	Next item in the memory node list
wait_queue_head_t	kswapd_wait	Wait queue for the *kswapd* pageout daemon (see the section "Periodic Reclaiming" in Chapter 17)
struct task_struct *	kswapd	Pointer to the process descriptor of the *kswapd* kernel thread
int	kswapd_max_order	Logarithmic size of free blocks to be created by *kswapd*

As usual, we are mostly concerned with the 80×86 architecture. IBM-compatible PCs use the Uniform Memory Access model (UMA), thus the NUMA support is not really required. However, even if NUMA support is not compiled in the kernel, Linux makes use of a single node that includes all system physical memory. Thus, the pgdat_list variable points to a list consisting of a single element—the node 0 descriptor—stored in the contig_page_data variable.

On the 80×86 architecture, grouping the physical memory in a single node might appear useless; however, this approach makes the memory handling code more portable, because the kernel can assume that the physical memory is partitioned in one or more nodes in all architectures.*

* We have another example of this kind of design choice: Linux uses four levels of Page Tables even when the hardware architecture defines just two levels (see the section "Paging in Linux" in Chapter 2).

Memory Zones

In an ideal computer architecture, a page frame is a memory storage unit that can be used for anything: storing kernel and user data, buffering disk data, and so on. Every kind of page of data can be stored in a page frame, without limitations.

However, real computer architectures have hardware constraints that may limit the way page frames can be used. In particular, the Linux kernel must deal with two hardware constraints of the 80×86 architecture:

- The Direct Memory Access (DMA) processors for old ISA buses have a strong limitation: they are able to address only the first 16 MB of RAM.
- In modern 32-bit computers with lots of RAM, the CPU cannot directly access all physical memory because the linear address space is too small.

To cope with these two limitations, Linux 2.6 partitions the physical memory of every memory node into three *zones*. In the 80×86 UMA architecture the zones are:

`ZONE_DMA`
: Contains page frames of memory below 16 MB

`ZONE_NORMAL`
: Contains page frames of memory at and above 16 MB and below 896 MB

`ZONE_HIGHMEM`
: Contains page frames of memory at and above 896 MB

The `ZONE_DMA` zone includes page frames that can be used by old ISA-based devices by means of the DMA. (The section "Direct Memory Access (DMA)" in Chapter 13 gives further details on DMA.)

The `ZONE_DMA` and `ZONE_NORMAL` zones include the "normal" page frames that can be directly accessed by the kernel through the linear mapping in the fourth gigabyte of the linear address space (see the section "Kernel Page Tables" in Chapter 2). Conversely, the `ZONE_HIGHMEM` zone includes page frames that cannot be directly accessed by the kernel through the linear mapping in the fourth gigabyte of linear address space (see the section "Kernel Mappings of High-Memory Page Frames" later in this chapter). The `ZONE_HIGHMEM` zone is always empty on 64-bit architectures.

Each memory zone has its own descriptor of type `zone`. Its fields are shown in Table 8-4.

Table 8-4. The fields of the zone descriptor

Type	Name	Description
`unsigned long`	`free_pages`	Number of free pages in the zone.
`unsigned long`	`pages_min`	Number of reserved pages of the zone (see the section "The Pool of Reserved Page Frames" later in this chapter).

Table 8-4. The fields of the zone descriptor (continued)

Type	Name	Description
unsigned long	pages_low	Low watermark for page frame reclaiming; also used by the zone allocator as a threshold value (see the section "The Zone Allocator" later in this chapter).
unsigned long	pages_high	High watermark for page frame reclaiming; also used by the zone allocator as a threshold value.
unsigned long []	lowmem_reserve	Specifies how many page frames in each zone must be reserved for handling low-on-memory critical situations.
struct per_cpu_pageset[]	pageset	Data structure used to implement special caches of single page frames (see the section "The Per-CPU Page Frame Cache" later in this chapter).
spinlock_t	lock	Spin lock protecting the descriptor.
struct free_area []	free_area	Identifies the blocks of free page frames in the zone (see the section "The Buddy System Algorithm" later in this chapter).
spinlock_t	lru_lock	Spin lock for the active and inactive lists.
struct list head	active_list	List of active pages in the zone (see Chapter 17).
struct list head	inactive_list	List of inactive pages in the zone (see Chapter 17).
unsigned long	nr_scan_active	Number of active pages to be scanned when reclaiming memory (see the section "Low On Memory Reclaiming" in Chapter 17).
unsigned long	nr_scan_inactive	Number of inactive pages to be scanned when reclaiming memory.
unsigned long	nr_active	Number of pages in the zone's active list.
unsigned long	nr_inactive	Number of pages in the zone's inactive list.
unsigned long	pages_scanned	Counter used when doing page frame reclaiming in the zone.
int	all_unreclaimable	Flag set when the zone is full of unreclaimable pages.
int	temp_priority	Temporary zone's priority (used when doing page frame reclaiming).
int	prev_priority	Zone's priority ranging between 12 and 0 (used by the page frame reclaiming algorithm, see the section "Low On Memory Reclaiming" in Chapter 17).
wait_queue_head_t *	wait_table	Hash table of wait queues of processes waiting for one of the pages of the zone.
unsigned long	wait_table_size	Size of the wait queue hash table.
unsigned long	wait_table_bits	Power-of-2 order of the size of the wait queue hash table array.
struct pglist_data *	zone_pgdat	Memory node (see the earlier section "Non-Uniform Memory Access (NUMA)").
struct page *	zone_mem_map	Pointer to first page descriptor of the zone.
unsigned long	zone_start_pfn	Index of the first page frame of the zone.
unsigned long	spanned_pages	Total size of zone in pages, including holes.

Table 8-4. The fields of the zone descriptor (continued)

Type	Name	Description
unsigned long	present_pages	Total size of zone in pages, excluding holes.
char *	name	Pointer to the conventional name of the zone: "DMA," "Normal," or "HighMem."

Many fields of the zone structure are used for page frame reclaiming and will be described in Chapter 17.

Each page descriptor has links to the memory node and to the zone inside the node that includes the corresponding page frame. To save space, these links are not stored as classical pointers; rather, they are encoded as indices stored in the high bits of the flags field. In fact, the number of flags that characterize a page frame is limited, thus it is always possible to reserve the most significant bits of the flags field to encode the proper memory node and zone number.* The page_zone( ) function receives as its parameter the address of a page descriptor; it reads the most significant bits of the flags field in the page descriptor, then it determines the address of the corresponding zone descriptor by looking in the zone_table array. This array is initialized at boot time with the addresses of all zone descriptors of all memory nodes.

When the kernel invokes a memory allocation function, it must specify the zones that contain the requested page frames. The kernel usually specifies which zones it's willing to use. For instance, if a page frame must be directly mapped in the fourth gigabyte of linear addresses but it is not going to be used for ISA DMA transfers, then the kernel requests a page frame either in ZONE_NORMAL or in ZONE_DMA. Of course, the page frame should be obtained from ZONE_DMA only if ZONE_NORMAL does not have free page frames. To specify the preferred zones in a memory allocation request, the kernel uses the zonelist data structure, which is an array of zone descriptor pointers.

The Pool of Reserved Page Frames

Memory allocation requests can be satisfied in two different ways. If enough free memory is available, the request can be satisfied immediately. Otherwise, some memory reclaiming must take place, and the kernel control path that made the request is blocked until additional memory has been freed.

However, some kernel control paths cannot be blocked while requesting memory—this happens, for instance, when handling an interrupt or when executing code inside a critical region. In these cases, a kernel control path should issue *atomic memory*

* The number of bits reserved for the indices depends on whether the kernel supports the NUMA model and on the size of the flags field. If NUMA is not supported, the flags field has two bits for the zone index and one bit—always set to zero—for the node index. On NUMA 32-bit architectures, flags has two bits for the zone index and six bits for the node number. Finally, on NUMA 64-bit architectures, the 64-bit flags field has 2 bits for the zone index and 10 bits for the node number.

allocation requests (using the GFP_ATOMIC flag; see the later section "The Zoned Page Frame Allocator"). An atomic request never blocks: if there are not enough free pages, the allocation simply fails.

Although there is no way to ensure that an atomic memory allocation request never fails, the kernel tries hard to minimize the likelihood of this unfortunate event. In order to do this, the kernel reserves a pool of page frames for atomic memory allocation requests to be used only on low-on-memory conditions.

The amount of the reserved memory (in kilobytes) is stored in the min_free_kbytes variable. Its initial value is set during kernel initialization and depends on the amount of physical memory that is directly mapped in the kernel's fourth gigabyte of linear addresses—that is, it depends on the number of page frames included in the ZONE_DMA and ZONE_NORMAL memory zones:

$$\textit{reserved pool size} = \lfloor \sqrt{16 \times \textit{directly mapped memory}} \rfloor \quad \text{(kilobytes)}$$

However, initially min_free_kbytes cannot be lower than 128 and greater than 65,536.*

The ZONE_DMA and ZONE_NORMAL memory zones contribute to the reserved memory with a number of page frames proportional to their relative sizes. For instance, if the ZONE_NORMAL zone is eight times bigger than ZONE_DMA, seven-eighths of the page frames will be taken from ZONE_NORMAL and one-eighth from ZONE_DMA.

The pages_min field of the zone descriptor stores the number of reserved page frames inside the zone. As we'll see in Chapter 17, this field plays also a role for the page frame reclaiming algorithm, together with the pages_low and pages_high fields. The pages_low field is always set to 5/4 of the value of pages_min, and pages_high is always set to 3/2 of the value of pages_min.

The Zoned Page Frame Allocator

The kernel subsystem that handles the memory allocation requests for groups of contiguous page frames is called the *zoned page frame allocator*. Its main components are shown in Figure 8-2.

The component named "zone allocator" receives the requests for allocation and deallocation of dynamic memory. In the case of allocation requests, the component searches a memory zone that includes a group of contiguous page frames that can satisfy the request (see the later section "The Zone Allocator"). Inside each zone, page frames are handled by a component named "buddy system" (see the later section "The Buddy System Algorithm"). To get better system performance, a small

* The amount of reserved memory can be changed later by the system administrator either by writing in the */proc/sys/vm/min_free_kbytes* file or by issuing a suitable sysctl() system call.

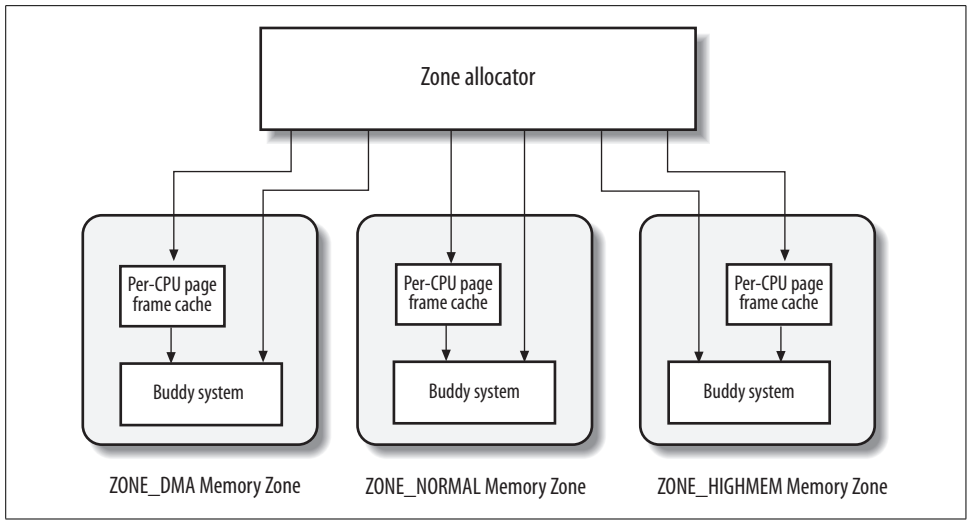


Figure 8-2. Components of the zoned page frame allocator

number of page frames are kept in cache to quickly satisfy the allocation requests for single page frames (see the later section "The Per-CPU Page Frame Cache").

Requesting and releasing page frames

Page frames can be requested by using six slightly different functions and macros. Unless otherwise stated, they return the linear address of the first allocated page or return `NULL` if the allocation failed.

`alloc_pages(gfp_mask, order)`
: Macro used to request 2^{order} contiguous page frames. It returns the address of the descriptor of the first allocated page frame or returns `NULL` if the allocation failed.

`alloc_page(gfp_mask)`
: Macro used to get a single page frame; it expands to:

```
alloc_pages(gfp_mask, 0)
```

: It returns the address of the descriptor of the allocated page frame or returns `NULL` if the allocation failed.

`__get_free_pages(gfp_mask, order)`
: Function that is similar to `alloc_pages()`, but it returns the linear address of the first allocated page.

`__get_free_page(gfp_mask)`
: Macro used to get a single page frame; it expands to:

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

`get_zeroed_page(gfp_mask)`
: Function used to obtain a page frame filled with zeros; it invokes:

```
alloc_pages(gfp_mask | __GFP_ZERO, 0)
```

and returns the linear address of the obtained page frame.

`__get_dma_pages(gfp_mask, order)`
: Macro used to get page frames suitable for DMA; it expands to:

```
__get_free_pages(gfp_mask | __GFP_DMA, order)
```

The parameter `gfp_mask` is a group of flags that specify how to look for free page frames. The flags that can be used in `gfp_mask` are shown in Table 8-5.

Table 8-5. Flag used to request page frames

Flag	Description
`__GFP_DMA`	The page frame must belong to the `ZONE_DMA` memory zone. Equivalent to `GFP_DMA`.
`__GFP_HIGHMEM`	The page frame may belong to the `ZONE_HIGHMEM` memory zone.
`__GFP_WAIT`	The kernel is allowed to block the current process waiting for free page frames.
`__GFP_HIGH`	The kernel is allowed to access the pool of reserved page frames.
`__GFP_IO`	The kernel is allowed to perform I/O transfers on low memory pages in order to free page frames.
`__GFP_FS`	If clear, the kernel is not allowed to perform filesystem-dependent operations.
`__GFP_COLD`	The requested page frames may be "cold" (see the later section "The Per-CPU Page Frame Cache").
`__GFP_NOWARN`	A memory allocation failure will not produce a warning message.
`__GFP_REPEAT`	The kernel keeps retrying the memory allocation until it succeeds.
`__GFP_NOFAIL`	Same as `__GFP_REPEAT`.
`__GFP_NORETRY`	Do not retry a failed memory allocation.
`__GFP_NO_GROW`	The slab allocator does not allow a slab cache to be enlarged (see the later section "The Slab Allocator").
`__GFP_COMP`	The page frame belongs to an extended page (see the section "Extended Paging" in Chapter 2).
`__GFP_ZERO`	The page frame returned, if any, must be filled with zeros.

In practice, Linux uses the predefined combinations of flag values shown in Table 8-6; the group name is what you'll encounter as the argument of the six page frame allocation functions.

Table 8-6. Groups of flag values used to request page frames

Group name	Corresponding flags
`GFP_ATOMIC`	`__GFP_HIGH`
`GFP_NOIO`	`__GFP_WAIT`
`GFP_NOFS`	`__GFP_WAIT | __GFP_IO`
`GFP_KERNEL`	`__GFP_WAIT | __GFP_IO | __GFP_FS`
`GFP_USER`	`__GFP_WAIT | __GFP_IO | __GFP_FS`
`GFP_HIGHUSER`	`__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM`

The __GFP_DMA and __GFP_HIGHMEM flags are called *zone modifiers*; they specify the zones searched by the kernel while looking for free page frames. The node_zonelists field of the contig_page_data node descriptor is an array of lists of zone descriptors representing the *fallback zones*: for each setting of the zone modifiers, the corresponding list includes the memory zones that could be used to satisfy the memory allocation request in case the original zone is short on page frames. In the 80×86 UMA architecture, the fallback zones are the following:

- If the __GFP_DMA flag is set, page frames can be taken only from the ZONE_DMA memory zone.
- Otherwise, if the __GFP_HIGHMEM flag is *not* set, page frames can be taken only from the ZONE_NORMAL and the ZONE_DMA memory zones, in order of preference.
- Otherwise (the __GFP_HIGHMEM flag is set), page frames can be taken from ZONE_HIGHMEM, ZONE_NORMAL, and ZONE_DMA memory zones, in order of preference.

Page frames can be released through each of the following four functions and macros:

`__free_pages(page, order)`
: This function checks the page descriptor pointed to by page; if the page frame is not reserved (i.e., if the PG_reserved flag is equal to 0), it decreases the count field of the descriptor. If count becomes 0, it assumes that 2^{order} contiguous page frames starting from the one corresponding to page are no longer used. In this case, the function releases the page frames as explained in the later section "The Zone Allocator."

`free_pages(addr, order)`
: This function is similar to __free_pages(), but it receives as an argument the linear address addr of the first page frame to be released.

`__free_page(page)`
: This macro releases the page frame having the descriptor pointed to by page; it expands to:

```
__free_pages(page, 0)
```

`free_page(addr)`
: This macro releases the page frame having the linear address addr; it expands to:

```
free_pages(addr, 0)
```

Kernel Mappings of High-Memory Page Frames

The linear address that corresponds to the end of the directly mapped physical memory, and thus to the beginning of the high memory, is stored in the high_memory variable, which is set to 896 MB. Page frames above the 896 MB boundary are not generally mapped in the fourth gigabyte of the kernel linear address spaces, so the kernel is unable to directly access them. This implies that each page allocator function that returns the linear address of the assigned page frame doesn't work for *high-memory* page frames, that is, for page frames in the ZONE_HIGHMEM memory zone.

For instance, suppose that the kernel invoked `__get_free_pages(GFP_HIGHMEM,0)` to allocate a page frame in high memory. If the allocator assigned a page frame in high memory, `__get_free_pages()` cannot return its linear address because it doesn't exist; thus, the function returns `NULL`. In turn, the kernel cannot use the page frame; even worse, the page frame cannot be released because the kernel has lost track of it.

This problem does not exist on 64-bit hardware platforms, because the available linear address space is much larger than the amount of RAM that can be installed—in short, the `ZONE_HIGHMEM` zone of these architectures is always empty. On 32-bit platforms such as the 80×86 architecture, however, Linux designers had to find some way to allow the kernel to exploit all the available RAM, up to the 64 GB supported by PAE. The approach adopted is the following:

- The allocation of high-memory page frames is done only through the `alloc_pages()` function and its `alloc_page()` shortcut. These functions do not return the linear address of the first allocated page frame, because if the page frame belongs to the high memory, such linear address simply does not exist. Instead, the functions return the linear address of the page descriptor of the first allocated page frame. These linear addresses always exist, because all page descriptors are allocated in low memory once and forever during the kernel initialization.
- Page frames in high memory that do not have a linear address cannot be accessed by the kernel. Therefore, part of the last 128 MB of the kernel linear address space is dedicated to mapping high-memory page frames. Of course, this kind of mapping is temporary, otherwise only 128 MB of high memory would be accessible. Instead, by recycling linear addresses the whole high memory can be accessed, although at different times.

The kernel uses three different mechanisms to map page frames in high memory; they are called *permanent kernel mapping*, *temporary kernel mapping*, and *noncontiguous memory allocation*. In this section, we'll cover the first two techniques; the third one is discussed in the section "Noncontiguous Memory Area Management" later in this chapter.

Establishing a permanent kernel mapping may block the current process; this happens when no free Page Table entries exist that can be used as "windows" on the page frames in high memory. Thus, a permanent kernel mapping cannot be established in interrupt handlers and deferrable functions. Conversely, establishing a temporary kernel mapping never requires blocking the current process; its drawback, however, is that very few temporary kernel mappings can be established at the same time.

A kernel control path that uses a temporary kernel mapping must ensure that no other kernel control path is using the same mapping. This implies that the kernel control path can never block, otherwise another kernel control path might use the same window to map some other high memory page.

Of course, none of these techniques allow addressing the whole RAM simultaneously. After all, less than 128 MB of linear address space are left for mapping the high memory, while PAE supports systems having up to 64 GB of RAM.

Permanent kernel mappings

Permanent kernel mappings allow the kernel to establish long-lasting mappings of high-memory page frames into the kernel address space. They use a dedicated Page Table in the master kernel page tables. The `pkmap_page_table` variable stores the address of this Page Table, while the `LAST_PKMAP` macro yields the number of entries. As usual, the Page Table includes either 512 or 1,024 entries, according to whether PAE is enabled or disabled (see the section "The Physical Address Extension (PAE) Paging Mechanism" in Chapter 2); thus, the kernel can access at most 2 or 4 MB of high memory at once.

The Page Table maps the linear addresses starting from `PKMAP_BASE`. The `pkmap_count` array includes `LAST_PKMAP` counters, one for each entry of the `pkmap_page_table` Page Table. We distinguish three cases:

The counter is 0
: The corresponding Page Table entry does not map any high-memory page frame and is usable.

The counter is 1
: The corresponding Page Table entry does not map any high-memory page frame, but it cannot be used because the corresponding TLB entry has not been flushed since its last usage.

The counter is n (greater than 1)
: The corresponding Page Table entry maps a high-memory page frame, which is used by exactly $n-1$ kernel components.

To keep track of the association between high memory page frames and linear addresses induced by permanent kernel mappings, the kernel makes use of the `page_address_htable` hash table. This table contains one `page_address_map` data structure for each page frame in high memory that is currently mapped. In turn, this data structure contains a pointer to the page descriptor and the linear address assigned to the page frame.

The `page_address()` function returns the linear address associated with the page frame, or `NULL` if the page frame is in high memory and is not mapped. This function, which receives as its parameter a page descriptor pointer `page`, distinguishes two cases:

1. If the page frame is not in high memory (`PG_highmem` flag clear), the linear address always exists and is obtained by computing the page frame index, converting it into a physical address, and finally deriving the linear address corresponding to the physical address. This is accomplished by the following code:

```
__va((unsigned long)(page - mem_map) << 12)
```

2. If the page frame is in high memory (PG_highmem flag set), the function looks into the page_address_htable hash table. If the page frame is found in the hash table, page_address() returns its linear address, otherwise it returns NULL.

The kmap() function establishes a permanent kernel mapping. It is essentially equivalent to the following code:

```
void * kmap(struct page * page)
{
    if (!PageHighMem(page))
        return page_address(page);
    return kmap_high(page);
}
```

The kmap_high() function is invoked if the page frame really belongs to high memory. The function is essentially equivalent to the following code:

```
void * kmap_high(struct page * page)
{
    unsigned long vaddr;
    spin_lock(&kmap_lock);
    vaddr = (unsigned long) page_address(page);
    if (!vaddr)
        vaddr = map_new_virtual(page);
    pkmap_count[(vaddr-PKMAP_BASE) >> PAGE_SHIFT]++;
    spin_unlock(&kmap_lock);
    return (void *) vaddr;
}
```

The function gets the kmap_lock spin lock to protect the Page Table against concurrent accesses in multiprocessor systems. Notice that there is no need to disable the interrupts, because kmap() cannot be invoked by interrupt handlers and deferrable functions. Next, the kmap_high() function checks whether the page frame is already mapped by invoking page_address(). If not, the function invokes map_new_virtual() to insert the page frame physical address into an entry of pkmap_page_table and to add an element to the page_address_htable hash table. Then kmap_high() increases the counter corresponding to the linear address of the page frame to take into account the new kernel component that invoked this function. Finally, kmap_high() releases the kmap_lock spin lock and returns the linear address that maps the page frame.

The map_new_virtual() function essentially executes two nested loops:

```
    for (;;) {
        int count;
        DECLARE_WAITQUEUE(wait, current);
        for (count = LAST_PKMAP; count > 0; --count) {
            last_pkmap_nr = (last_pkmap_nr + 1) & (LAST_PKMAP - 1);
            if (!last_pkmap_nr) {
                flush_all_zero_pkmaps();
                count = LAST_PKMAP;
            }
            if (!pkmap_count[last_pkmap_nr]) {
```

```
                unsigned long vaddr = PKMAP_BASE +
                                      (last_pkmap_nr << PAGE_SHIFT);
                set_pte(&(pkmap_page_table[last_pkmap_nr]),
                        mk_pte(page, __pgprot(0x63)));
                pkmap_count[last_pkmap_nr] = 1;
                set_page_address(page, (void *) vaddr);
                return vaddr;
            }
        }
        current->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue(&pkmap_map_wait, &wait);
        spin_unlock(&kmap_lock);
        schedule();
        remove_wait_queue(&pkmap_map_wait, &wait);
        spin_lock(&kmap_lock);
        if (page_address(page))
            return (unsigned long) page_address(page);
    }
```

In the inner loop, the function scans all counters in `pkmap_count` until it finds a null value. The large `if` block runs when an unused entry is found in `pkmap_count`. That block determines the linear address corresponding to the entry, creates an entry for it in the `pkmap_page_table` Page Table, sets the count to 1 because the entry is now used, invokes `set_page_address()` to insert a new element in the `page_address_htable` hash table, and returns the linear address.

The function starts where it left off last time, cycling through the `pkmap_count` array. It does this by preserving in a variable named `last_pkmap_nr` the index of the last used entry in the `pkmap_page_table` Page Table. Thus, the search starts from where it was left in the last invocation of the `map_new_virtual()` function.

When the last counter in `pkmap_count` is reached, the search restarts from the counter at index 0. Before continuing, however, `map_new_virtual()` invokes the `flush_all_zero_pkmaps()` function, which starts another scan of the counters, looking for those that have the value 1. Each counter that has a value of 1 denotes an entry in `pkmap_page_table` that is free but cannot be used because the corresponding TLB entry has not yet been flushed. `flush_all_zero_pkmaps()` resets their counters to zero, deletes the corresponding elements from the `page_address_htable` hash table, and issues TLB flushes on all entries of `pkmap_page_table`.

If the inner loop cannot find a null counter in `pkmap_count`, the `map_new_virtual()` function blocks the current process until some other process releases an entry of the `pkmap_page_table` Page Table. This is achieved by inserting `current` in the `pkmap_map_wait` wait queue, setting the `current` state to `TASK_UNINTERRUPTIBLE`, and invoking `schedule()` to relinquish the CPU. Once the process is awakened, the function checks whether another process has mapped the page by invoking `page_address()`; if no other process has mapped the page yet, the inner loop is restarted.

The kunmap() function destroys a permanent kernel mapping established previously by kmap(). If the page is really in the high memory zone, it invokes the kunmap_high() function, which is essentially equivalent to the following code:

```
void kunmap_high(struct page * page)
{
    spin_lock(&kmap_lock);
    if ((--pkmap_count[((unsigned long)page_address(page)
                        -PKMAP_BASE)>>PAGE_SHIFT]) == 1)
        if (waitqueue_active(&pkmap_map_wait))
            wake_up(&pkmap_map_wait);
    spin_unlock(&kmap_lock);
}
```

The expression within the brackets computes the index into the pkmap_count array from the page's linear address. The counter is decreased and compared to 1. A successful comparison indicates that no process is using the page. The function can finally wake up processes in the wait queue filled by map_new_virtual(), if any.

Temporary kernel mappings

Temporary kernel mappings are simpler to implement than permanent kernel mappings; moreover, they can be used inside interrupt handlers and deferrable functions, because requesting a temporary kernel mapping never blocks the current process.

Every page frame in high memory can be mapped through a *window* in the kernel address space—namely, a Page Table entry that is reserved for this purpose. The number of windows reserved for temporary kernel mappings is quite small.

Each CPU has its own set of 13 windows, represented by the enum km_type data structure. Each symbol defined in this data structure—such as KM_BOUNCE_READ, KM_USER0, or KM_PTE0—identifies the linear address of a window.

The kernel must ensure that the same window is never used by two kernel control paths at the same time. Thus, each symbol in the km_type structure is dedicated to one kernel component and is named after the component. The last symbol, KM_TYPE_NR, does not represent a linear address by itself, but yields the number of different windows usable by every CPU.

Each symbol in km_type, except the last one, is an index of a fix-mapped linear address (see the section "Fix-Mapped Linear Addresses" in Chapter 2). The enum fixed_addresses data structure includes the symbols FIX_KMAP_BEGIN and FIX_KMAP_END; the latter is assigned to the index FIX_KMAP_BEGIN + (KM_TYPE_NR * NR_CPUS) - 1. In this manner, there are KM_TYPE_NR fix-mapped linear addresses for each CPU in the system. Furthermore, the kernel initializes the kmap_pte variable with the address of the Page Table entry corresponding to the fix_to_virt(FIX_KMAP_BEGIN) linear address.

To establish a temporary kernel mapping, the kernel invokes the kmap_atomic() function, which is essentially equivalent to the following code:

```
void * kmap_atomic(struct page * page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;

    current_thread_info()->preempt_count++;
    if (!PageHighMem(page))
        return page_address(page);
    idx = type + KM_TYPE_NR * smp_processor_id();
    vaddr = fix_to_virt(FIX_KMAP_BEGIN + idx);
    set_pte(kmap_pte-idx, mk_pte(page, 0x063));
    __flush_tlb_single(vaddr);
    return (void *) vaddr;
}
```

The type argument and the CPU identifier retrieved through smp_processor_id() specify what fix-mapped linear address has to be used to map the request page. The function returns the linear address of the page frame if it doesn't belong to high memory; otherwise, it sets up the Page Table entry corresponding to the fix-mapped linear address with the page's physical address and the bits Present, Accessed, Read/Write, and Dirty. Finally, the function flushes the proper TLB entry and returns the linear address.

To destroy a temporary kernel mapping, the kernel uses the kunmap_atomic() function. In the 80×86 architecture, this function decreases the preempt_count of the current process; thus, if the kernel control path was preemptable right before requiring a temporary kernel mapping, it will be preemptable again after it has destroyed the same mapping. Moreover, kunmap_atomic() checks whether the TIF_NEED_RESCHED flag of current is set and, if so, invokes schedule().

The Buddy System Algorithm

The kernel must establish a robust and efficient strategy for allocating groups of contiguous page frames. In doing so, it must deal with a well-known memory management problem called *external fragmentation*: frequent requests and releases of groups of contiguous page frames of different sizes may lead to a situation in which several small blocks of free page frames are "scattered" inside blocks of allocated page frames. As a result, it may become impossible to allocate a large block of contiguous page frames, even if there are enough free pages to satisfy the request.

There are essentially two ways to avoid external fragmentation:

- Use the paging circuitry to map groups of noncontiguous free page frames into intervals of contiguous linear addresses.

- Develop a suitable technique to keep track of the existing blocks of free contiguous page frames, avoiding as much as possible the need to split up a large free block to satisfy a request for a smaller one.

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

- In some cases, contiguous page frames are really necessary, because contiguous linear addresses are not sufficient to satisfy the request. A typical example is a memory request for buffers to be assigned to a DMA processor (see Chapter 13). Because most DMAs ignore the paging circuitry and access the address bus directly while transferring several disk sectors in a single I/O operation, the buffers requested must be located in contiguous page frames.
- Even if contiguous page frame allocation is not strictly necessary, it offers the big advantage of leaving the kernel paging tables unchanged. What's wrong with modifying the Page Tables? As we know from Chapter 2, frequent Page Table modifications lead to higher average memory access times, because they make the CPU flush the contents of the translation lookaside buffers.
- Large chunks of contiguous physical memory can be accessed by the kernel through 4 MB pages. This reduces the translation lookaside buffers misses, thus significantly speeding up the average memory access time (see the section "Translation Lookaside Buffers (TLB)" in Chapter 2).

The technique adopted by Linux to solve the external fragmentation problem is based on the well-known *buddy system* algorithm. All free page frames are grouped into 11 lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 contiguous page frames, respectively. The largest request of 1024 page frames corresponds to a chunk of 4 MB of contiguous RAM. The physical address of the first page frame of a block is a multiple of the group size—for example, the initial address of a 16-page-frame block is a multiple of 16×2^{12} ($2^{12} = 4{,}096$, which is the regular page size).

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

Assume there is a request for a group of 256 contiguous page frames (i.e., one megabyte). The algorithm checks first to see whether a free block in the 256-page-frame list exists. If there is no such block, the algorithm looks for the next larger block—a free block in the 512-page-frame list. If such a block exists, the kernel allocates 256 of the 512 page frames to satisfy the request and inserts the remaining 256 page frames into the list of free 256-page-frame blocks. If there is no free 512-page block, the kernel then looks for the next larger block (i.e., a free 1024-page-frame block). If such a block exists, it allocates 256 of the 1024 page frames to satisfy the request, inserts the first 512 of the remaining 768 page frames into the list of free 512-page-frame blocks, and inserts the last 256 page frames into the list of free 256-page-frame blocks. If the list of 1024-page-frame blocks is empty, the algorithm gives up and signals an error condition.

The reverse operation, releasing blocks of page frames, gives rise to the name of this algorithm. The kernel attempts to merge pairs of free buddy blocks of size *b* together into a single block of size 2*b*. Two blocks are considered buddies if:

- Both blocks have the same size, say *b*.
- They are located in contiguous physical addresses.
- The physical address of the first page frame of the first block is a multiple of $2 \times b \times 2^{12}$.

The algorithm is iterative; if it succeeds in merging released blocks, it doubles *b* and tries again so as to create even bigger blocks.

Data structures

Linux 2.6 uses a different buddy system for each zone. Thus, in the 80×86 architecture, there are 3 buddy systems: the first handles the page frames suitable for ISA DMA, the second handles the "normal" page frames, and the third handles the high-memory page frames. Each buddy system relies on the following main data structures:

- The `mem_map` array introduced previously. Actually, each zone is concerned with a subset of the `mem_map` elements. The first element in the subset and its number of elements are specified, respectively, by the `zone_mem_map` and `size` fields of the zone descriptor.
- An array consisting of eleven elements of type `free_area`, one element for each group size. The array is stored in the `free_area` field of the zone descriptor.

Let us consider the k^{th} element of the `free_area` array in the zone descriptor, which identifies all the free blocks of size 2^k. The `free_list` field of this element is the head of a doubly linked circular list that collects the page descriptors associated with the free blocks of 2^k pages. More precisely, this list includes the page descriptors of the starting page frame of every block of 2^k free page frames; the pointers to the adjacent elements in the list are stored in the `lru` field of the page descriptor.*

Besides the head of the list, the k^{th} element of the `free_area` array includes also the field `nr_free`, which specifies the number of free blocks of size 2^k pages. Of course, if there are no blocks of 2^k free page frames, `nr_free` is equal to 0 and the `free_list` list is empty (both pointers of `free_list` point to the `free_list` field itself).

Finally, the `private` field of the descriptor of the first page in a block of 2^k free pages stores the order of the block, that is, the number *k*. Thanks to this field, when a block of pages is freed, the kernel can determine whether the buddy of the block is also free and, if so, it can coalesce the two blocks in a single block of 2^{k+1} pages. It

* As we'll see later, the `lru` field of the page descriptor can be used with other meanings when the page is not free.

should be noted that up to Linux 2.6.10, the kernel used 10 arrays of flags to encode this information.

Allocating a block

The `__rmqueue()` function is used to find a free block in a zone. The function takes two arguments: the address of the zone descriptor, and `order`, which denotes the logarithm of the size of the requested block of free pages (0 for a one-page block, 1 for a two-page block, and so forth). If the page frames are successfully allocated, the `__rmqueue()` function returns the address of the page descriptor of the first allocated page frame. Otherwise, the function returns `NULL`.

The `__rmqueue()` function assumes that the caller has already disabled local interrupts and acquired the `zone->lock` spin lock, which protects the data structures of the buddy system. It performs a cyclic search through each list for an available block (denoted by an entry that doesn't point to the entry itself), starting with the list for the requested `order` and continuing if necessary to larger orders:

```
struct free_area *area;
unsigned int current_order;

for (current_order=order; current_order<11; ++current_order) {
    area = zone->free_area + current_order;
    if (!list_empty(&area->free_list))
        goto block_found;
}
return NULL;
```

If the loop terminates, no suitable free block has been found, so `__rmqueue()` returns a `NULL` value. Otherwise, a suitable free block has been found; in this case, the descriptor of its first page frame is removed from the list and the value of `free_pages` in the zone descriptor is decreased:

```
block_found:
    page = list_entry(area->free_list.next, struct page, lru);
    list_del(&page->lru);
    ClearPagePrivate(page);
    page->private = 0;
    area->nr_free--;
    zone->free_pages -= 1UL << order;
```

If the block found comes from a list of size `curr_order` greater than the requested size `order`, a `while` cycle is executed. The rationale behind these lines of codes is as follows: when it becomes necessary to use a block of 2^k page frames to satisfy a request for 2^h page frames ($h < k$), the program allocates the first 2^h page frames and iteratively reassigns the last $2^k - 2^h$ page frames to the `free_area` lists that have indexes between h and k:

```
    size = 1 << curr_order;
    while (curr_order > order) {
        area--;
        curr_order--;
```

```
            size >>= 1;
            buddy = page + size;
            /* insert buddy as first element in the list */
            list_add(&buddy->lru, &area->free_list);
            area->nr_free++;
            buddy->private = curr_order;
            SetPagePrivate(buddy);
        }
        return page;
```

Because the `__rmqueue()` function has found a suitable free block, it returns the address `page` of the page descriptor associated with the first allocated page frame.

Freeing a block

The `__free_pages_bulk()` function implements the buddy system strategy for freeing page frames. It uses three basic input parameters:*

`page`
: The address of the descriptor of the first page frame included in the block to be released

`zone`
: The address of the zone descriptor

`order`
: The logarithmic size of the block

The function assumes that the caller has already disabled local interrupts and acquired the `zone->lock` spin lock, which protects the data structure of the buddy system. `__free_pages_bulk()` starts by declaring and initializing a few local variables:

```
struct page * base = zone->zone_mem_map;
unsigned long buddy_idx, page_idx = page - base;
struct page * buddy, * coalesced;
int order_size = 1 << order;
```

The `page_idx` local variable contains the index of the first page frame in the block with respect to the first page frame of the zone.

The `order_size` local variable is used to increase the counter of free page frames in the zone:

```
zone->free_pages += order_size;
```

The function now performs a cycle executed at most 10–`order` times, once for each possibility for merging a block with its buddy. The function starts with the smallest-sized block and moves up to the top size:

```
while (order < 10) {
    buddy_idx = page_idx ^ (1 << order);
```

* For performance reasons, this inline function also uses another parameter; its value, however, can be determined by the three basic parameters shown in the text.

```
        buddy = base + buddy_idx;
        if (!page_is_buddy(buddy, order))
            break;
        list_del(&buddy->lru);
        zone->free_area[order].nr_free--;
        ClearPagePrivate(buddy);
        buddy->private = 0;
        page_idx &= buddy_idx;
        order++;
    }
```

In the body of the loop, the function looks for the index `buddy_idx` of the block, which is buddy to the one having the page descriptor index `page_idx`. It turns out that this index can be easily computed as:

```
buddy_idx = page_idx ^ (1 << order);
```

In fact, an Exclusive OR (XOR) using the `(1<<order)` mask switches the value of the `order`-th bit of `page_idx`. Therefore, if the bit was previously zero, `buddy_idx` is equal to `page_idx + order_size`; conversely, if the bit was previously one, `buddy_idx` is equal to `page_idx - order_size`.

Once the buddy block index is known, the page descriptor of the buddy block can be easily obtained as:

```
buddy = base + buddy_idx;
```

Now the function invokes `page_is_buddy()` to check if `buddy` describes the first page of a block of `order_size` free page frames.

```
int page_is_buddy(struct page *page, int order)
{
    if (PagePrivate(buddy) && page->private == order &&
          !PageReserved(buddy) && page_count(page) ==0)
        return 1;
    return 0;
}
```

As you see, the buddy's first page must be free (`_count` field equal to -1), it must belong to the dynamic memory (`PG_reserved` bit clear), its `private` field must be meaningful (`PG_private` bit set), and finally the `private` field must store the order of the block being freed.

If all these conditions are met, the buddy block is free and the function removes the buddy block from the list of free blocks of order `order`, and performs one more iteration looking for buddy blocks twice as big.

If at least one of the conditions in `page_is_buddy()` is not met, the function breaks out of the cycle, because the free block obtained cannot be merged further with other free blocks. The function inserts it in the proper list and updates the `private` field of the first page frame with the order of the block size:

```
coalesced = base + page_idx;
coalesced->private = order;
```

```
SetPagePrivate(coalesced);
list_add(&coalesced->lru, &zone->free_area[order].free_list);
zone->free_area[order].nr_free++;
```

The Per-CPU Page Frame Cache

As we will see later in this chapter, the kernel often requests and releases single page frames. To boost system performance, each memory zone defines a *per-CPU page frame cache*. Each per-CPU cache includes some pre-allocated page frames to be used for single memory requests issued by the local CPU.

Actually, there are two caches for each memory zone and for each CPU: a *hot cache*, which stores page frames whose contents are likely to be included in the CPU's hardware cache, and a *cold cache*.

Taking a page frame from the hot cache is beneficial for system performance if either the kernel or a User Mode process will write into the page frame right after the allocation. In fact, every access to a memory cell of the page frame will result in a line of the hardware cache being "stolen" from another page frame—unless, of course, the hardware cache already includes a line that maps the cell of the "hot" page frame just accessed.

Conversely, taking a page frame from the cold cache is convenient if the page frame is going to be filled with a DMA operation. In this case, the CPU is not involved and no line of the hardware cache will be modified. Taking the page frame from the cold cache preserves the reserve of hot page frames for the other kinds of memory allocation requests.

The main data structure implementing the per-CPU page frame cache is an array of `per_cpu_pageset` data structures stored in the `pageset` field of the memory zone descriptor. The array includes one element for each CPU; this element, in turn, consists of two `per_cpu_pages` descriptors, one for the hot cache and the other for the cold cache. The fields of the `per_cpu_pages` descriptor are listed in Table 8-7.

Table 8-7. The fields of the per_cpu_pages descriptor

Type	Name	Description
`int`	`count`	Number of pages frame in the cache
`int`	`low`	Low watermark for cache replenishing
`int`	`high`	High watermark for cache depletion
`int`	`batch`	Number of page frames to be added or subtracted from the cache
`struct list_head`	`list`	List of descriptors of the page frames included in the cache

The kernel monitors the size of the both the hot and cold caches by using two watermarks: if the number of page frames falls below the `low` watermark, the kernel replenishes the proper cache by allocating `batch` single page frames from the buddy

system; otherwise, if the number of page frames rises above the high watermark, the kernel releases to the buddy system batch page frames in the cache. The values of batch, low, and high essentially depend on the number of page frames included in the memory zone.

Allocating page frames through the per-CPU page frame caches

The buffered_rmqueue() function allocates page frames in a given memory zone. It makes use of the per-CPU page frame caches to handle single page frame requests.

The parameters are the address of the memory zone descriptor, the order of the memory allocation request order, and the allocation flags gfp_flags. If the __GFP_COLD flag is set in gfp_flags, the page frame should be taken from the cold cache, otherwise it should be taken from the hot cache (this flag is meaningful only for single page frame requests). The function essentially executes the following operations:

1. If order is not equal to 0, the per-CPU page frame cache cannot be used: the function jumps to step 4.
2. Checks whether the memory zone's local per-CPU cache identified by the value of the __GFP_COLD flag has to be replenished (the count field of the per_cpu_pages descriptor is lower than or equal to the low field). In this case, it executes the following substeps:
   a. Allocates batch single page frames from the buddy system by repeatedly invoking the __rmqueue() function.
   b. Inserts the descriptors of the allocated page frames in the cache's list.
   c. Updates the value of count by adding the number of page frames actually allocated.
3. If count is positive, the function gets a page frame from the cache's list, decreases count, and jumps to step 5. (Observe that a per-CPU page frame cache could be empty; this happens when the __rmqueue() function invoked in step 2a fails to allocate any page frames.)
4. Here, the memory request has not yet been satisfied, either because the request spans several contiguous page frames, or because the selected page frame cache is empty. Invokes the __rmqueue() function to allocate the requested page frames from the buddy system.
5. If the memory request has been satisfied, the function initializes the page descriptor of the (first) page frame: clears some flags, sets the private field to zero, and sets the page frame reference counter to one. Moreover, if the __GPF_ZERO flag in gfp_flags is set, it fills the allocated memory area with zeros.
6. Returns the page descriptor address of the (first) page frame, or NULL if the memory allocation request failed.

Releasing page frames to the per-CPU page frame caches

In order to release a single page frame to a per-CPU page frame cache, the kernel makes use of the `free_hot_page()` and `free_cold_page()` functions. Both of them are simple wrappers for the `free_hot_cold_page()` function, which receives as its parameters the descriptor address `page` of the page frame to be released and a `cold` flag specifying either the hot cache or the cold cache.

The `free_hot_cold_page()` function executes the following operations:

1. Gets from the `page->flags` field the address of the memory zone descriptor including the page frame (see the earlier section "Non-Uniform Memory Access (NUMA)").
2. Gets the address of the `per_cpu_pages` descriptor of the zone's cache selected by the `cold` flag.
3. Checks whether the cache should be depleted: if `count` is higher than or equal to `high`, invokes the `free_pages_bulk()` function, passing to it the zone descriptor, the number of page frames to be released (`batch` field), the address of the cache's list, and the number zero (for 0-order page frames). In turn, the latter function invokes repeatedly the `__free_pages_bulk()` function to releases the specified number of page frames—taken from the cache's list—to the buddy system of the memory zone.
4. Adds the page frame to be released to the cache's list, and increases the `count` field.

It should be noted that in the current version of the Linux 2.6 kernel, no page frame is ever released to the cold cache: the kernel always assumes the freed page frame is hot with respect to the hardware cache. Of course, this does not mean that the cold cache is empty: the cache is replenished by `buffered_rmqueue()` when the low watermark has been reached.

The Zone Allocator

The *zone allocator* is the frontend of the kernel page frame allocator. This component must locate a memory zone that includes a number of free page frames large enough to satisfy the memory request. This task is not as simple as it could appear at a first glance, because the zone allocator must satisfy several goals:

- It should protect the pool of reserved page frames (see the earlier section "The Pool of Reserved Page Frames").
- It should trigger the page frame reclaiming algorithm (see Chapter 17) when memory is scarce and blocking the current process is allowed; once some page frames have been freed, the zone allocator will retry the allocation.
- It should preserve the small, precious `ZONE_DMA` memory zone, if possible. For instance, the zone allocator should be somewhat reluctant to assign page frames

in the ZONE_DMA memory zone if the request was for ZONE_NORMAL or ZONE_HIGHMEM page frames.

We have seen in the earlier section "The Zoned Page Frame Allocator" that every request for a group of contiguous page frames is eventually handled by executing the alloc_pages macro. This macro, in turn, ends up invoking the __alloc_pages() function, which is the core of the zone allocator. It receives three parameters:

gfp_mask
: The flags specified in the memory allocation request (see earlier Table 8-5)

order
: The logarithmic size of the group of contiguous page frames to be allocated

zonelist
: Pointer to a zonelist data structure describing, in order of preference, the memory zones suitable for the memory allocation

The __alloc_pages() function scans every memory zone included in the zonelist data structure. The code that does this looks like the following:

```
for (i = 0; (z=zonelist->zones[i]) != NULL; i++) {
    if (zone_watermark_ok(z, order, ...)) {
        page = buffered_rmqueue(z, order, gfp_mask);
        if (page)
            return page;
    }
}
```

For each memory zone, the function compares the number of free page frames with a threshold value that depends on the memory allocation flags, on the type of current process, and on how many times the zone has already been checked by the function. In fact, if free memory is scarce, every memory zone is typically scanned several times, each time with lower threshold on the minimal amount of free memory required for the allocation. The previous block of code is thus replicated several times—with minor variations—in the body of the __alloc_pages() function. The buffered_rmqueue() function has been described already in the earlier section "The Per-CPU Page Frame Cache:" it returns the page descriptor of the first allocated page frame, or NULL if the memory zone does not include a group of contiguous page frames of the requested size.

The zone_watermark_ok() auxiliary function receives several parameters, which determine a threshold min on the number of free page frames in the memory zone. In particular, the function returns the value 1 if the following two conditions are met:

1. Besides the page frames to be allocated, there are at least min free page frames in the memory zone, not including the page frames in the low-on-memory reserve (lowmem_reserve field of the zone descriptor).
2. Besides the page frames to be allocated, there are at least $min/2^k$ free page frames in blocks of order at least k, for each k between 1 and the order of the

allocation. Therefore, if order is greater than zero, there must be at least min/2 free page frames in blocks of size at least 2; if order is greater than one, there must be at least min/4 free page frames in blocks of size at least 4; and so on.

The value of the threshold min is determined by zone_watermark_ok() as follows:

- The base value is passed as a parameter of the function and can be one of the pages_min, pages_low, and pages_high zone's watermarks (see the section "The Pool of Reserved Page Frames" earlier in this chapter).
- The base value is divided by two if the gfp_high flag passed as parameter is set. Usually, this flag is equal to one if the __GFP_HIGHMEM flag is set in the gfp_mask, that is, if the page frames can be allocated from high memory.
- The threshold value is further reduced by one-fourth if the can_try_harder flag passed as parameter is set. This flag is usually equal to one if either the __GFP_WAIT flag is set in gfp_mask, or if the current process is a real-time process and the memory allocation is done in process context (outside of interrupt handlers and deferrable functions).

The __alloc_pages() function essentially executes the following steps:

1. Performs a first scanning of the memory zones (see the block of code shown earlier). In this first scan, the min threshold value is set to z->pages_low, where z points to the zone descriptor being analyzed (the can_try_harder and gfp_high parameters are set to zero).
2. If the function did not terminate in the previous step, there is not much free memory left: the function awakens the *kswapd* kernel threads to start reclaiming page frames asynchronously (see Chapter 17).
3. Performs a second scanning of the memory zones, passing as base threshold the value z->pages_min. As explained previously, the actual threshold is determined also by the can_try_harder and gfp_high flags. This step is nearly identical to step 1, except that the function is using a lower threshold.
4. If the function did not terminate in the previous step, the system is definitely low on memory. If the kernel control path that issued the memory allocation request is not an interrupt handler or a deferrable function and it is trying to reclaim page frames (either the PF_MEMALLOC flag or the PF_MEMDIE flag of current is set), the function then performs a third scanning of the memory zones, trying to allocate the page frames ignoring the low-on-memory thresholds—that is, without invoking zone_watermark_ok(). This is the only case where the kernel control path is allowed to deplete the low-on-memory reserve of pages specified by the lowmem_reserve field of the zone descriptor. In fact, in this case the kernel control path that issued the memory request is ultimately trying to free page frames, thus it should get what it has requested, if at all possible. If no memory zone includes enough page frames, the function returns NULL to notify the caller of the failure.

5. Here, the invoking kernel control path is not trying to reclaim memory. If the __GFP_WAIT flag of gfp_mask is not set, the function returns NULL to notify the kernel control path of the memory allocation failure: in this case, there is no way to satisfy the request without blocking the current process.
6. Here the current process can be blocked: invokes cond_resched() to check whether some other process needs the CPU.
7. Sets the PF_MEMALLOC flag of current, to denote the fact that the process is ready to perform memory reclaiming.
8. Stores in current->reclaim_state a pointer to a reclaim_state structure. This structure includes just one field, reclaimed_slab, initialized to zero (we'll see how this field is used in the section "Interfacing the Slab Allocator with the Zoned Page Frame Allocator" later in this chapter).
9. Invokes try_to_free_pages() to look for some page frames to be reclaimed (see the section "Low On Memory Reclaiming" in Chapter 17). The latter function may block the current process. Once that function returns, __alloc_pages() resets the PF_MEMALLOC flag of current and invokes once more cond_resched().
10. If the previous step has freed some page frames, the function performs yet another scanning of the memory zones equal to the one performed in step 3. If the memory allocation request cannot be satisfied, the function determines whether it should continue scanning the memory zone: if the __GFP_NORETRY flag is clear and either the memory allocation request spans up to eight page frames, or one of the __GFP_REPEAT and __GFP_NOFAIL flags is set, the function invokes blk_congestion_wait() to put the process asleep for awhile (see Chapter 14), and it jumps back to step 6. Otherwise, the function returns NULL to notify the caller that the memory allocation failed.
11. If no page frame has been freed in step 9, the kernel is in deep trouble, because free memory is dangerously low and it was not possible to reclaim any page frame. Perhaps the time has come to take a crucial decision. If the kernel control path is allowed to perform the filesystem-dependent operations needed to kill a process (the __GFP_FS flag in gfp_mask is set) and the __GFP_NORETRY flag is clear, performs the following substeps:
    a. Scans once again the memory zones with a threshold value equal to z->pages_high.
    b. Invokes out_of_memory() to start freeing some memory by killing a victim process (see "The Out of Memory Killer" in Chapter 17).
    c. Jumps back to step 1.

    Because the watermark used in step 11a is much higher than the watermarks used in the previous scannings, that step is likely to fail. Actually, step 11a succeeds only if another kernel control path is already killing a process to reclaim its memory. Thus, step 11a avoids that two innocent processes are killed instead of one.

Releasing a group of page frames

The zone allocator also takes care of releasing page frames; thankfully, releasing memory is a lot easier than allocating it.

All kernel macros and functions that release page frames—described in the earlier section "The Zoned Page Frame Allocator"—rely on the `__free_pages()` function. It receives as its parameters the address of the page descriptor of the first page frame to be released (`page`), and the logarithmic size of the group of contiguous page frames to be released (`order`). The function executes the following steps:

1. Checks that the first page frame really belongs to dynamic memory (its `PG_reserved` flag is cleared); if not, terminates.
2. Decreases the `page->_count` usage counter; if it is still greater than or equal to zero, terminates.
3. If `order` is equal to zero, the function invokes `free_hot_page()` to release the page frame to the per-CPU hot cache of the proper memory zone (see the earlier section "The Per-CPU Page Frame Cache").
4. If `order` is greater than zero, it adds the page frames in a local list and invokes the `free_pages_bulk()` function to release them to the buddy system of the proper memory zone (see step 3 in the description of `free_hot_cold_page()` in the earlier section "The Per-CPU Page Frame Cache").

Memory Area Management

This section deals with *memory areas*—that is, with sequences of memory cells having contiguous physical addresses and an arbitrary length.

The buddy system algorithm adopts the page frame as the basic memory area. This is fine for dealing with relatively large memory requests, but how are we going to deal with requests for small memory areas, say a few tens or hundreds of bytes?

Clearly, it would be quite wasteful to allocate a full page frame to store a few bytes. A better approach instead consists of introducing new data structures that describe how small memory areas are allocated within the same page frame. In doing so, we introduce a new problem called *internal fragmentation*. It is caused by a mismatch between the size of the memory request and the size of the memory area allocated to satisfy the request.

A classical solution (adopted by early Linux versions) consists of providing memory areas whose sizes are geometrically distributed; in other words, the size depends on a power of 2 rather than on the size of the data to be stored. In this way, no matter what the memory request size is, we can ensure that the internal fragmentation is always smaller than 50 percent. Following this approach, the kernel creates 13 geometrically distributed lists of free memory areas whose sizes range from 32 to 131,072 bytes. The buddy system is invoked both to obtain additional page frames

needed to store new memory areas and, conversely, to release page frames that no longer contain memory areas. A dynamic list is used to keep track of the free memory areas contained in each page frame.

The Slab Allocator

Running a memory area allocation algorithm on top of the buddy algorithm is not particularly efficient. A better algorithm is derived from the *slab allocator* schema that was adopted for the first time in the Sun Microsystems Solaris 2.4 operating system. It is based on the following premises:

- The type of data to be stored may affect how memory areas are allocated; for instance, when allocating a page frame to a User Mode process, the kernel invokes the `get_zeroed_page( )` function, which fills the page with zeros.

  The concept of a slab allocator expands upon this idea and views the memory areas as *objects* consisting of both a set of data structures and a couple of functions or methods called the *constructor* and *destructor*. The former initializes the memory area while the latter deinitializes it.

  To avoid initializing objects repeatedly, the slab allocator does not discard the objects that have been allocated and then released but instead saves them in memory. When a new object is then requested, it can be taken from memory without having to be reinitialized.
- The kernel functions tend to request memory areas of the same type repeatedly. For instance, whenever the kernel creates a new process, it allocates memory areas for some fixed size tables such as the process descriptor, the open file object, and so on (see Chapter 3). When a process terminates, the memory areas used to contain these tables can be reused. Because processes are created and destroyed quite frequently, without the slab allocator, the kernel wastes time allocating and deallocating the page frames containing the same memory areas repeatedly; the slab allocator allows them to be saved in a cache and reused quickly.
- Requests for memory areas can be classified according to their frequency. Requests of a particular size that are expected to occur frequently can be handled most efficiently by creating a set of special-purpose objects that have the right size, thus avoiding internal fragmentation. Meanwhile, sizes that are rarely encountered can be handled through an allocation scheme based on objects in a series of geometrically distributed sizes (such as the power-of-2 sizes used in early Linux versions), even if this approach leads to internal fragmentation.
- There is another subtle bonus in introducing objects whose sizes are not geometrically distributed: the initial addresses of the data structures are less prone to be concentrated on physical addresses whose values are a power of 2. This, in turn, leads to better performance by the processor hardware cache.

- Hardware cache performance creates an additional reason for limiting calls to the buddy system allocator as much as possible. Every call to a buddy system function "dirties" the hardware cache, thus increasing the average memory access time. The impact of a kernel function on the hardware cache is called the function *footprint*; it is defined as the percentage of cache overwritten by the function when it terminates. Clearly, large footprints lead to a slower execution of the code executed right after the kernel function, because the hardware cache is by now filled with useless information.

The slab allocator groups objects into *caches*. Each cache is a "store" of objects of the same type. For instance, when a file is opened, the memory area needed to store the corresponding "open file" object is taken from a slab allocator cache named *filp* (for "file pointer").

The area of main memory that contains a cache is divided into *slabs*; each slab consists of one or more contiguous page frames that contain both allocated and free objects (see Figure 8-3).

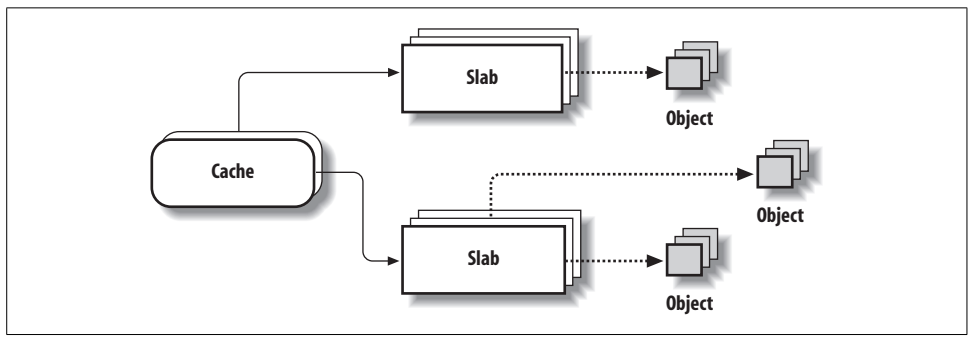


Figure 8-3. The slab allocator components

As we'll see in Chapter 17, the kernel periodically scans the caches and releases the page frames corresponding to empty slabs.

Cache Descriptor

Each cache is described by a structure of type `kmem_cache_t` (which is equivalent to the type `struct kmem_cache_s`), whose fields are listed in Table 8-8. We omitted from the table several fields used for collecting statistical information and for debugging.

Table 8-8. The fields of the kmem_cache_t descriptor

Type	Name	Description
`struct array_cache * []`	`array`	Per-CPU array of pointers to local caches of free objects (see the section "Local Caches of Free Slab Objects" later in this chapter).
`unsigned int`	`batchcount`	Number of objects to be transferred in bulk to or from the local caches.
`unsigned int`	`limit`	Maximum number of free objects in the local caches. This is tunable.

Table 8-8. The fields of the kmem_cache_t descriptor (continued)

Type	Name	Description
struct kmem_list3	lists	See next table.
unsigned int	objsize	Size of the objects included in the cache.
unsigned int	flags	Set of flags that describes permanent properties of the cache.
unsigned int	num	Number of objects packed into a single slab. (All slabs of the cache have the same size.)
unsigned int	free_limit	Upper limit of free objects in the whole slab cache.
spinlock_t	spinlock	Cache spin lock.
unsigned int	gfporder	Logarithm of the number of contiguous page frames included in a single slab.
unsigned int	gfpflags	Set of flags passed to the buddy system function when allocating page frames.
size_t	colour	Number of colors for the slabs (see the section "Slab Coloring" later in this chapter).
unsigned int	colour_off	Basic alignment offset in the slabs.
unsigned int	colour_next	Color to use for the next allocated slab.
kmem_cache_t *	slabp_cache	Pointer to the general slab cache containing the slab descriptors (NULL if internal slab descriptors are used; see next section).
unsigned int	slab_size	The size of a single slab.
unsigned int	dflags	Set of flags that describe dynamic properties of the cache.
void *	ctor	Pointer to constructor method associated with the cache.
void *	dtor	Pointer to destructor method associated with the cache.
const char *	name	Character array storing the name of the cache.
struct list_head	next	Pointers for the doubly linked list of cache descriptors.

The lists field of the kmem_cache_t descriptor, in turn, is a structure whose fields are listed in Table 8-9.

Table 8-9. The fields of the kmem_list3 structure

Type	Name	Description
struct list_head	slabs_partial	Doubly linked circular list of slab descriptors with both free and non-free objects
struct list_head	slabs_full	Doubly linked circular list of slab descriptors with no free objects
struct list_head	slabs_free	Doubly linked circular list of slab descriptors with free objects only
unsigned long	free_objects	Number of free objects in the cache
int	free_touched	Used by the slab allocator's page reclaiming algorithm (see Chapter 17)
unsigned long	next_reap	Used by the slab allocator's page reclaiming algorithm (see Chapter 17)

Table 8-9. The fields of the kmem_list3 structure (continued)

Type	Name	Description
`struct array_cache *`	`shared`	Pointer to a local cache shared by all CPUs (see the later section "Local Caches of Free Slab Objects")

Slab Descriptor

Each slab of a cache has its own descriptor of type `slab` illustrated in Table 8-10.

Table 8-10. The fields of the slab descriptor

Type	Name	Description
`struct list_head`	`list`	Pointers for one of the three doubly linked list of slab descriptors (either the `slabs_full`, `slabs_partial`, or `slabs_free` list in the `kmem_list3` structure of the cache descriptor)
`unsigned long`	`colouroff`	Offset of the first object in the slab (see the section "Slab Coloring" later in this chapter)
`void *`	`s_mem`	Address of first object (either allocated or free) in the slab
`unsigned int`	`inuse`	Number of objects in the slab that are currently used (not free)
`unsigned int`	`free`	Index of next free object in the slab, or `BUFCTL_END` if there are no free objects left (see the section "Object Descriptor" later in this chapter)

Slab descriptors can be stored in two possible places:

External slab descriptor
: Stored outside the slab, in one of the general caches not suitable for ISA DMA pointed to by `cache_sizes` (see the next section).

Internal slab descriptor
: Stored inside the slab, at the beginning of the first page frame assigned to the slab.

The slab allocator chooses the second solution when the size of the objects is smaller than 512MB or when internal fragmentation leaves enough space for the slab descriptor and the object descriptors (as described later)—inside the slab. The `CFLGS_OFF_SLAB` flag in the `flags` field of the cache descriptor is set to one if the slab descriptor is stored outside the slab; it is set to zero otherwise.

Figure 8-4 illustrates the major relationships between cache and slab descriptors. Full slabs, partially full slabs, and free slabs are linked in different lists.

General and Specific Caches

Caches are divided into two types: general and specific. *General caches* are used only by the slab allocator for its own purposes, while *specific caches* are used by the remaining parts of the kernel.

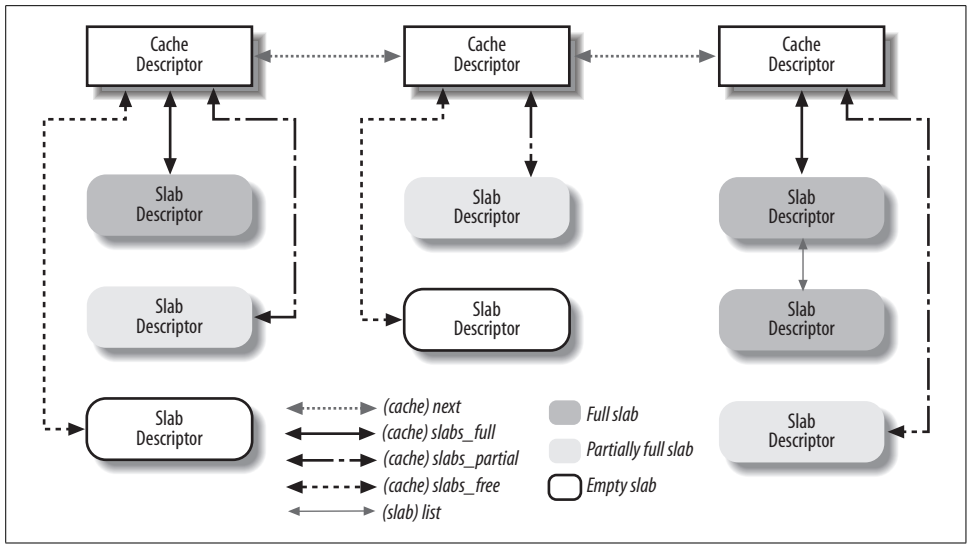


Figure 8-4. Relationship between cache and slab descriptors

The general caches are:

- A first cache called *kmem_cache* whose objects are the cache descriptors of the remaining caches used by the kernel. The `cache_cache` variable contains the descriptor of this special cache.
- Several additional caches contain general purpose memory areas. The range of the memory area sizes typically includes 13 geometrically distributed sizes. A table called `malloc_sizes` (whose elements are of type `cache_sizes`) points to 26 cache descriptors associated with memory areas of size 32, 64, 128, 256, 512, 1,024, 2,048, 4,096, 8,192, 16,384, 32,768, 65,536, and 131,072 bytes. For each size, there are two caches: one suitable for ISA DMA allocations and the other for normal allocations.

The `kmem_cache_init()` function is invoked during system initialization to set up the general caches.

Specific caches are created by the `kmem_cache_create()` function. Depending on the parameters, the function first determines the best way to handle the new cache (for instance, whether to include the slab descriptor inside or outside of the slab). It then allocates a cache descriptor for the new cache from the `cache_cache` general cache and inserts the descriptor in the `cache_chain` list of cache descriptors (the insertion is done after having acquired the `cache_chain_sem` semaphore that protects the list from concurrent accesses).

It is also possible to destroy a cache and remove it from the cache_chain list by invoking kmem_cache_destroy( ). This function is mostly useful to modules that create their own caches when loaded and destroy them when unloaded. To avoid wasting memory space, the kernel must destroy all slabs before destroying the cache itself. The kmem_cache_shrink( ) function destroys all the slabs in a cache by invoking slab_destroy( ) iteratively (see the later section "Releasing a Slab from a Cache").

The names of all general and specific caches can be obtained at runtime by reading */proc/slabinfo*; this file also specifies the number of free objects and the number of allocated objects in each cache.

Interfacing the Slab Allocator with the Zoned Page Frame Allocator

When the slab allocator creates a new slab, it relies on the zoned page frame allocator to obtain a group of free contiguous page frames. For this purpose, it invokes the kmem_getpages( ) function, which is essentially equivalent, on a UMA system, to the following code fragment:

```
void * kmem_getpages(kmem_cache_t *cachep, int flags)
{
    struct page *page;
    int i;

    flags |= cachep->gfpflags;
    page = alloc_pages(flags, cachep->gfporder);
    if (!page)
        return NULL;
    i = (1 << cache->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        atomic_add(i, &slab_reclaim_pages);
    while (i--)
            SetPageSlab(page++);
    return page_address(page);
}
```

The two parameters have the following meaning:

cachep
: Points to the cache descriptor of the cache that needs additional page frames (the number of required page frames is determined by the order in the cachep->gfporder field).

flags
: Specifies how the page frame is requested (see the section "The Zoned Page Frame Allocator" earlier in this chapter). This set of flags is combined with the specific cache allocation flags stored in the gfpflags field of the cache descriptor.

The size of the memory allocation request is specified by the gfporder field of the cache descriptor, which encodes the size of a slab in the cache.* If the slab cache has been created with the SLAB_RECLAIM_ACCOUNT flag set, the page frames assigned to the slabs are accounted for as reclaimable pages when the kernel checks whether there is enough memory to satisfy some User Mode requests. The function also sets the PG_slab flag in the page descriptors of the allocated page frames.

In the reverse operation, page frames assigned to a slab can be released (see the section "Releasing a Slab from a Cache" later in this chapter) by invoking the kmem_freepages() function:

```
void kmem_freepages(kmem_cache_t *cachep, void *addr)
{
    unsigned long i = (1<<cachep->gfporder);
    struct page *page = virt_to_page(addr);

    if (current->reclaim_state)
        current->reclaim_state->reclaimed_slab += i;
    while (i--)
        ClearPageSlab(page++);
    free_pages((unsigned long) addr, cachep->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        atomic_sub(1<<cachep->gfporder, &slab_reclaim_pages);
}
```

The function releases the page frames, starting from the one having the linear address addr, that had been allocated to the slab of the cache identified by cachep. If the current process is performing memory reclaiming (current->reclaim_state field not NULL), the reclaimed_slab field of the reclaim_state structure is properly increased, so that the pages just freed can be accounted for by the page frame reclaiming algorithm (see the section "Low On Memory Reclaiming" in Chapter 17). Moreover, if the SLAB_RECLAIM_ACCOUNT flag is set (see above), the slab_reclaim_pages variable is properly decreased.

Allocating a Slab to a Cache

A newly created cache does not contain a slab and therefore does not contain any free objects. New slabs are assigned to a cache only when both of the following are true:

- A request has been issued to allocate a new object.
- The cache does not include a free object.

* Notice that it is not possible to allocate page frames from the ZONE_HIGHMEM memory zone, because the kmem_getpages() function returns the linear address yielded by the page_address() function; as explained in the section "Kernel Mappings of High-Memory Page Frames" earlier in this chapter, this function returns NULL for unmapped high-memory page frames.

Under these circumstances, the slab allocator assigns a new slab to the cache by invoking `cache_grow()`. This function calls `kmem_getpages()` to obtain from the zoned page frame allocator the group of page frames needed to store a single slab; it then calls `alloc_slabmgmt()` to get a new slab descriptor. If the `CFLGS_OFF_SLAB` flag of the cache descriptor is set, the slab descriptor is allocated from the general cache pointed to by the `slabp_cache` field of the cache descriptor; otherwise, the slab descriptor is allocated in the first page frame of the slab.

The kernel must be able to determine, given a page frame, whether it is used by the slab allocator and, if so, to derive quickly the addresses of the corresponding cache and slab descriptors. Therefore, `cache_grow()` scans all page descriptors of the page frames assigned to the new slab, and loads the `next` and `prev` subfields of the `lru` fields in the page descriptors with the addresses of, respectively, the cache descriptor and the slab descriptor. This works correctly because the `lru` field is used by functions of the buddy system only when the page frame is free, while page frames handled by the slab allocator functions have the `PG_slab` flag set and are not free as far as the buddy system is concerned.* The opposite question—given a slab in a cache, which are the page frames that implement it?—can be answered by using the `s_mem` field of the slab descriptor and the `gfporder` field (the size of a slab) of the cache descriptor.

Next, `cache_grow()` calls `cache_init_objs()`, which applies the constructor method (if defined) to all the objects contained in the new slab.

Finally, `cache_grow()` calls `list_add_tail()` to add the newly obtained slab descriptor `*slabp` at the end of the fully free slab list of the cache descriptor `*cachep`, and updates the counter of free objects in the cache:

```
list_add_tail(&slabp->list, &cachep->lists->slabs_free);
cachep->lists->free_objects += cachep->num;
```

Releasing a Slab from a Cache

Slabs can be destroyed in two cases:

- There are too many free objects in the slab cache (see the later section "Releasing a Slab from a Cache").
- A timer function invoked periodically determines that there are fully unused slabs that can be released (see Chapter 17).

In both cases, the `slab_destroy()` function is invoked to destroy a slab and release the corresponding page frames to the zoned page frame allocator:

```
void slab_destroy(kmem_cache_t *cachep, slab_t *slabp)
{
    if (cachep->dtor) {
        int i;
```

* As we'll in Chapter 17, the `lru` field is also used by the page frame reclaiming algorithm.

```
            for (i = 0; i < cachep->num; i++) {
                void* objp = slabp->s_mem+cachep->objsize*i;
                (cachep->dtor)(objp, cachep, 0);
            }
        }
        kmem_freepages(cachep, slabp->s_mem - slabp->colouroff);
        if (cachep->flags & CFLGS_OFF_SLAB)
            kmem_cache_free(cachep->slabp_cache, slabp);
    }
```

The function checks whether the cache has a destructor method for its objects (the `dtor` field is not `NULL`), in which case it applies the destructor to all the objects in the slab; the `objp` local variable keeps track of the currently examined object. Next, it calls `kmem_freepages()`, which returns all the contiguous page frames used by the slab to the buddy system. Finally, if the slab descriptor is stored outside of the slab, the function releases it from the cache of slab descriptors.

Actually, the function is slightly more complicated. For example, a slab cache can be created with the `SLAB_DESTROY_BY_RCU` flag, which means that slabs should be released in a deferred way by registering a callback with the `call_rcu()` function (see the section "Read-Copy Update (RCU)" in Chapter 5). The callback function, in turn, invokes `kmem_freepages()` and, possibly, the `kmem_cache_free()`, as in the main case shown above.

Object Descriptor

Each object has a short descriptor of type `kmem_bufctl_t`. Object descriptors are stored in an array placed right after the corresponding slab descriptor. Thus, like the slab descriptors themselves, the object descriptors of a slab can be stored in two possible ways that are illustrated in Figure 8-5.

External object descriptors
: Stored outside the slab, in the general cache pointed to by the `slabp_cache` field of the cache descriptor. The size of the memory area, and thus the particular general cache used to store object descriptors, depends on the number of objects stored in the slab (`num` field of the cache descriptor).

Internal object descriptors
: Stored inside the slab, right before the objects they describe.

The first object descriptor in the array describes the first object in the slab, and so on. An object descriptor is simply an unsigned short integer, which is meaningful only when the object is free. It contains the index of the next free object in the slab, thus implementing a simple list of free objects inside the slab. The object descriptor of the last element in the free object list is marked by the conventional value `BUFCTL_END` (`0xffff`).

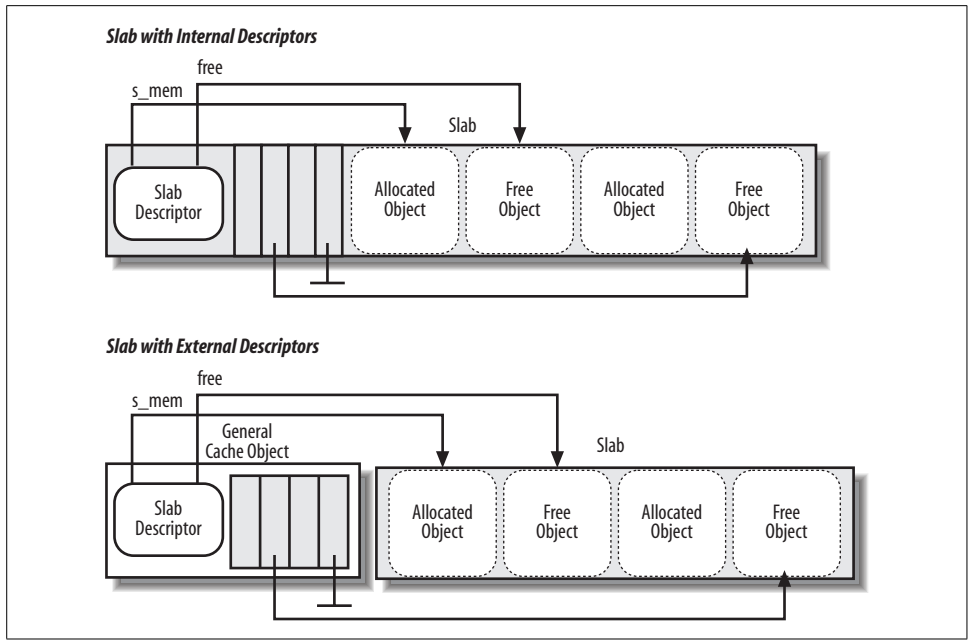


Figure 8-5. Relationships between slab and object descriptors

Aligning Objects in Memory

The objects managed by the slab allocator are *aligned* in memory—that is, they are stored in memory cells whose initial physical addresses are multiples of a given constant, which is usually a power of 2. This constant is called the *alignment factor*.

The largest alignment factor allowed by the slab allocator is 4,096—the page frame size. This means that objects can be aligned by referring to either their physical addresses or their linear addresses. In both cases, only the 12 least significant bits of the address may be altered by the alignment.

Usually, microcomputers access memory cells more quickly if their physical addresses are aligned with respect to the word size (that is, to the width of the internal memory bus of the computer). Thus, by default, the `kmem_cache_create( )` function aligns objects according to the word size specified by the `BYTES_PER_WORD` macro. For 80×86 processors, the macro yields the value 4 because the word is 32 bits long.

When creating a new slab cache, it's possible to specify that the objects included in it be aligned in the first-level hardware cache. To achieve this, the kernel sets the `SLAB_HWCACHE_ALIGN` cache descriptor flag. The `kmem_cache_create( )` function handles the request as follows:

- If the object's size is greater than half of a cache line, it is aligned in RAM to a multiple of `L1_CACHE_BYTES`—that is, at the beginning of the line.

- Otherwise, the object size is rounded up to a submultiple of `L1_CACHE_BYTES`; this ensures that a small object will never span across two cache lines.

Clearly, what the slab allocator is doing here is trading memory space for access time; it gets better cache performance by artificially increasing the object size, thus causing additional internal fragmentation.

Slab Coloring

We know from Chapter 2 that the same hardware cache line maps many different blocks of RAM. In this chapter, we have also seen that objects of the same size end up being stored at the same offset within a cache. Objects that have the same offset within different slabs will, with a relatively high probability, end up mapped in the same cache line. The cache hardware might therefore waste memory cycles transferring two objects from the same cache line back and forth to different RAM locations, while other cache lines go underutilized. The slab allocator tries to reduce this unpleasant cache behavior by a policy called *slab coloring*: different arbitrary values called *colors* are assigned to the slabs.

Before examining slab coloring, we have to look at the layout of objects in the cache. Let's consider a cache whose objects are aligned in RAM. This means that the object address must be a multiple of a given positive value, say *aln*. Even taking the alignment constraint into account, there are many possible ways to place objects inside the slab. The choices depend on decisions made for the following variables:

num
: Number of objects that can be stored in a slab (its value is in the `num` field of the cache descriptor).

osize
: Object size, including the alignment bytes.

dsize
: Slab descriptor size plus all object descriptors size, rounded up to the smallest multiple of the hardware cache line size. Its value is equal to 0 if the slab and object descriptors are stored outside of the slab.

free
: Number of unused bytes (bytes not assigned to any object) inside the slab.

The total length in bytes of a slab can then be expressed as:

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

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

The slab allocator takes advantage of the *free* unused bytes to color the slab. The term "color" is used simply to subdivide the slabs and allow the memory allocator to

spread objects out among different linear addresses. In this way, the kernel obtains the best possible performance from the microprocessor's hardware cache.

Slabs having different colors store the first object of the slab in different memory locations, while satisfying the alignment constraint. The number of available colors is *free/aln* (this value is stored in the `colour` field of the cache descriptor). Thus, the first color is denoted as 0 and the last one is denoted as (*free/aln*)–1. (As a particular case, if *free* is lower than *aln*, `colour` is set to 0, nevertheless all slabs use color 0, thus really the number of colors is one.)

If a slab is colored with color *col*, the offset of the first object (with respect to the slab initial address) is equal to *col* × *aln* + *dsize* bytes. Figure 8-6 illustrates how the placement of objects inside the slab depends on the slab color. Coloring essentially leads to moving some of the free area of the slab from the end to the beginning.

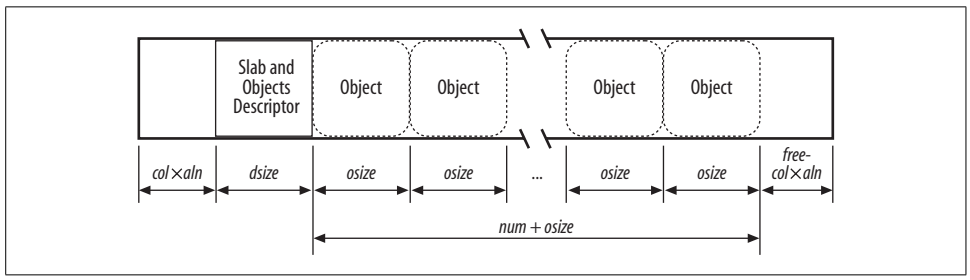


Figure 8-6. Slab with color col and alignment aln

Coloring works only when *free* is large enough. Clearly, if no alignment is required for the objects or if the number of unused bytes inside the slab is smaller than the required alignment (*free* < *aln*), the only possible slab coloring is the one that has the color 0—the one that assigns a zero offset to the first object.

The various colors are distributed equally among slabs of a given object type by storing the current color in a field of the cache descriptor called `colour_next`. The `cache_grow()` function assigns the color specified by `colour_next` to a new slab and then increases the value of this field. After reaching `colour`, it wraps around again to 0. In this way, each slab is created with a different color from the previous one, up to the maximum available colors. The `cache_grow()` function, moreover, gets the value *aln* from the `colour_off` field of the cache descriptor, computes *dsize* according to the number of objects inside the slab, and finally stores the value *col* × *aln* + *dsize* in the `colouroff` field of the slab descriptor.

Local Caches of Free Slab Objects

The Linux 2.6 implementation of the slab allocator for multiprocessor systems differs from that of the original Solaris 2.4. To reduce spin lock contention among processors and to make better use of the hardware caches, each cache of the slab

allocator includes a per-CPU data structure consisting of a small array of pointers to freed objects called the *slab local cache*. Most allocations and releases of slab objects affect the local cache only; the slab data structures get involved only when the local cache underflows or overflows. This technique is quite similar to the one illustrated in the section "The Per-CPU Page Frame Cache" earlier in this chapter.

The `array` field of the cache descriptor is an array of pointers to `array_cache` data structures, one element for each CPU in the system. Each `array_cache` data structure is a descriptor of the local cache of free objects, whose fields are illustrated in Table 8-11.

Table 8-11. The fields of the array_cache structure

Type	Name	Description
`unsigned int`	`avail`	Number of pointers to available objects in the local cache. The field also acts as the index of the first free slot in the cache.
`unsigned int`	`limit`	Size of the local cache—that is, the maximum number of pointers in the local cache.
`unsigned int`	`batchcount`	Chunk size for local cache refill or emptying.
`unsigned int`	`touched`	Flag set to 1 if the local cache has been recently used.

Notice that the local cache descriptor does not include the address of the local cache itself; in fact, the local cache is placed right after the descriptor. Of course, the local cache stores the pointers to the freed objects, not the object themselves, which are always placed inside the slabs of the cache.

When creating a new slab cache, the `kmem_cache_create()` function determines the size of the local caches (storing this value in the `limit` field of the cache descriptor), allocates them, and stores their pointers into the `array` field of the cache descriptor. The size depends on the size of the objects stored in the slab cache, and ranges from 1 for very large objects to 120 for small ones. Moreover, the initial value of the `batchcount` field, which is the number of objects added or removed in a chunk from a local cache, is initially set to half of the local cache size.*

In multiprocessor systems, slab caches for small objects also sport an additional local cache, whose address is stored in the `lists.shared` field of the cache descriptor. The *shared local cache* is, as the name suggests, shared among all CPUs, and it makes the task of migrating free objects from a local cache to another easier (see the following section). Its initial size is equal to eight times the value of the `batchcount` field.

* The system administrator can tune—for each cache—the size of the local caches and the value of the `batchcount` field by writing into the */proc/slabinfo* file.

Allocating a Slab Object

New objects may be obtained by invoking the `kmem_cache_alloc()` function. The parameter `cachep` points to the cache descriptor from which the new free object must be obtained, while the parameter `flag` represents the flags to be passed to the zoned page frame allocator functions, should all slabs of the cache be full.

The function is essentially equivalent to the following:

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
{
    unsigned long save_flags;
    void *objp;
    struct array_cache *ac;

    local_irq_save(save_flags);
    ac = cache_p->array[smp_processor_id()];
    if (ac->avail) {
        ac->touched = 1;
        objp = ((void **)(ac+1))[--ac->avail];
    } else
        objp = cache_alloc_refill(cachep, flags);
    local_irq_restore(save_flags);
    return objp;
}
```

The function tries first to retrieve a free object from the local cache. If there are free objects, the `avail` field contains the index in the local cache of the entry that points to the last freed object. Because the local cache array is stored right after the `ac` descriptor, `((void**)(ac+1))[--ac->avail]` gets the address of that free object and decreases the value of `ac->avail`. The `cache_alloc_refill()` function is invoked to repopulate the local cache and get a free object when there are no free objects in the local cache.

The `cache_alloc_refill()` function essentially performs the following steps:

1. Stores in the `ac` local variable the address of the local cache descriptor:

   ```
   ac = cachep->array[smp_processor_id()];
   ```

2. Gets the `cachep->spinlock`.
3. If the slab cache includes a shared local cache, and if the shared local cache includes some free objects, it refills the CPU's local cache by moving up to `ac->batchcount` pointers from the shared local cache. Then, it jumps to step 6.
4. Tries to fill the local cache with up to `ac->batchcount` pointers to free objects included in the slabs of the cache:
   a. Looks in the `slabs_partial` and `slabs_free` lists of the cache descriptor, and gets the address `slabp` of a slab descriptor whose corresponding slab is either partially filled or empty. If no such descriptor exists, the function goes to step 5.

b. For each free object in the slab, the function increases the `inuse` field of the slab descriptor, inserts the object's address in the local cache, and updates the `free` field so that it stores the index of the next free object in the slab:

```
slabp->inuse++;
((void**)(ac+1))[ac->avail++] =
                slabp->s_mem + slabp->free * cachep->obj_size;
slabp->free = ((kmem_bufctl_t*)(slabp+1))[slabp->free];
```

c. Inserts, if necessary, the depleted slab in the proper list, either the `slab_full` or the `slab_partial` list.

5. At this point, the number of pointers added to the local cache is stored in the `ac->avail` field: the function decreases the `free_objects` field of the `kmem_list3` structure of the same amount to specify that the objects are no longer free.
6. Releases the `cachep->spinlock`.
7. If the `ac->avail` field is now greater than 0 (some cache refilling took place), it sets the `ac->touched` field to 1 and returns the free object pointer that was last inserted in the local cache:

```
return ((void**)(ac+1))[--ac->avail];
```

8. Otherwise, no cache refilling took place: invokes `cache_grow()` to get a new slab, and thus new free objects.
9. If `cache_grow()` fails, it returns `NULL`; otherwise it goes back to step 1 to repeat the procedure.

Freeing a Slab Object

The `kmem_cache_free()` function releases an object previously allocated by the slab allocator to some kernel function. Its parameters are `cachep`, the address of the cache descriptor, and `objp`, the address of the object to be released:

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
{
    unsigned long flags;
    struct array_cache *ac;

    local_irq_save(flags);
    ac = cachep->array[smp_processor_id()];
    if (ac->avail == ac->limit)
        cache_flusharray(cachep, ac);
    ((void**)(ac+1))[ac->avail++] = objp;
    local_irq_restore(flags);
}
```

The function checks first whether the local cache has room for an additional pointer to a free object. If so, the pointer is added to the local cache and the function returns. Otherwise it first invokes `cache_flusharray()` to deplete the local cache and then adds the pointer to the local cache.

The cache_flusharray() function performs the following operations:

1. Acquires the cachep->spinlock spin lock.
2. If the slab cache includes a shared local cache, and if the shared local cache is not already full, it refills the shared local cache by moving up to ac->batchcount pointers from the CPU's local cache. Then, it jumps to step 4.
3. Invokes the free_block() function to give back to the slab allocator up to ac->batchcount objects currently included in the local cache. For each object at address objp, the function executes the following steps:
   a. Increases the lists.free_objects field of the cache descriptor.
   b. Determines the address of the slab descriptor containing the object:

      ```
      slabp = (struct slab *)(virt_to_page(objp)->lru.prev);
      ```

      (Remember that the lru.prev field of the descriptor of the slab page points to the corresponding slab descriptor.)
   c. Removes the slab descriptor from its slab cache list (either cachep->lists.slabs_partial or cachep->lists.slabs_full).
   d. Computes the index of the object inside the slab:

      ```
      objnr = (objp - slabp->s_mem) / cachep->objsize;
      ```

   e. Stores in the object descriptor the current value of the slabp->free, and puts in slabp->free the index of the object (the last released object will be the first object to be allocated again):

      ```
      ((kmem_bufctl_t *)(slabp+1))[objnr] = slabp->free;
      slabp->free = objnr;
      ```

   f. Decreases the slabp->inuse field.
   g. If slabp->inuse is equal to zero—all objects in the slab are free—and the number of free objects in the whole slab cache (cachep->lists.free_objects) is greater than the limit stored in the cachep->free_limit field, then the function releases the slab's page frame(s) to the zoned page frame allocator:

      ```
      cachep->lists.free_objects -= cachep->num;
      slab_destroy(cachep, slabp);
      ```

      The value stored in the cachep->free_limit field is usually equal to cachep->num + (1+*N*) × cachep->batchcount, where *N* denotes the number of CPUs of the system.
   h. Otherwise, if slab->inuse is equal to zero but the number of free objects in the whole slab cache is less than cachep->free_limit, it inserts the slab descriptor in the cachep->lists.slabs_free list.
   i. Finally, if slab->inuse is greater than zero, the slab is partially filled, so the function inserts the slab descriptor in the cachep->lists.slabs_partial list.
4. Releases the cachep->spinlock spin lock.

5. Updates the `avail` field of the local cache descriptor by subtracting the number of objects moved to the shared local cache or released to the slab allocator.
6. Moves all valid pointers in the local cache at the beginning of the local cache's array. This step is necessary because the first object pointers have been removed from the local cache, thus the remaining ones must be moved up.

General Purpose Objects

As stated earlier in the section "The Buddy System Algorithm," infrequent requests for memory areas are handled through a group of general caches whose objects have geometrically distributed sizes ranging from a minimum of 32 to a maximum of 131,072 bytes.

Objects of this type are obtained by invoking the `kmalloc( )` function, which is essentially equivalent to the following code fragment:

```
void * kmalloc(size_t size, int flags)
{
    struct cache_sizes *csizep = malloc_sizes;
    kmem_cache_t * cachep;
    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        if (flags & __GFP_DMA)
            cachep = csizep->cs_dmacachep;
        else
            cachep = csizep->cs_cachep;
        return kmem_cache_alloc(cachep, flags);
    }
    return NULL;
}
```

The function uses the `malloc_sizes` table to locate the nearest power-of-2 size to the requested size. It then calls `kmem_cache_alloc( )` to allocate the object, passing to it either the cache descriptor for the page frames usable for ISA DMA or the cache descriptor for the "normal" page frames, depending on whether the caller specified the `__GFP_DMA` flag.

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

```
void kfree(const void *objp)
{
    kmem_cache_t * c;
    unsigned long flags;
    if (!objp)
        return;
    local_irq_save(flags);
    c = (kmem_cache_t *)(virt_to_page(objp)->lru.next);
    kmem_cache_free(c, (void *)objp);
    local_irq_restore(flags);
}
```

The proper cache descriptor is identified by reading the `lru.next` subfield of the descriptor of the first page frame containing the memory area. The memory area is released by invoking `kmem_cache_free( )`.

Memory Pools

Memory pools are a new feature of Linux 2.6. Basically, a memory pool allows a kernel component—such as the block device subsystem—to allocate some dynamic memory to be used only in low-on-memory emergencies.

Memory pools should not be confused with the reserved page frames described in the earlier section "The Pool of Reserved Page Frames." In fact, those page frames can be used only to satisfy atomic memory allocation requests issued by interrupt handlers or inside critical regions. Instead, a memory pool is a reserve of dynamic memory that can be used only by a specific kernel component, namely the "owner" of the pool. The owner does not normally use the reserve; however, if dynamic memory becomes so scarce that all usual memory allocation requests are doomed to fail, the kernel component can invoke, as a last resort, special memory pool functions that dip in the reserve and get the memory needed. Thus, creating a memory pool is similar to keeping a reserve of canned foods on hand and using a can opener only when no fresh food is available.

Often, a memory pool is stacked over the slab allocator—that is, it is used to keep a reserve of slab objects. Generally speaking, however, a memory pool can be used to allocate every kind of dynamic memory, from whole page frames to small memory areas allocated with `kmalloc()`. Therefore, we will generically refer to the memory units handled by a memory pool as "memory elements."

A memory pool is described by a `mempool_t` object, whose fields are shown in Table 8-12.

Table 8-12. The fields of the mempool_t object

Type	Name	Description
`spinlock_t`	`lock`	Spin lock protecting the object fields
`int`	`min_nr`	Maximum number of elements in the memory pool
`int`	`curr_nr`	Current number of elements in the memory pool
`void **`	`elements`	Pointer to an array of pointers to the reserved elements
`void *`	`pool_data`	Private data available to the pool's owner
`mempool_alloc_t *`	`alloc`	Method to allocate an element
`mempool_free_t *`	`free`	Method to free an element
`wait_queue_head_t`	`wait`	Wait queue used when the memory pool is empty

The `min_nr` field stores the initial number of elements in the memory pool. In other words, the value stored in this field represents the number of memory elements that

the owner of the memory pool is sure to obtain from the memory allocator. The curr_nr field, which is always lower than or equal to min_nr, stores the number of memory elements currently included in the memory pool. The memory elements themselves are referenced by an array of pointers, whose address is stored in the elements field.

The alloc and free methods interface with the underlying memory allocator to get and release a memory element, respectively. Both methods may be custom functions provided by the kernel component that owns the memory pool.

When the memory elements are slab objects, the alloc and free methods are commonly implemented by the mempool_alloc_slab() and mempool_free_slab() functions, which just invoke the kmem_cache_alloc() and kmem_cache_free() functions, respectively. In this case, the pool_data field of the mempool_t object stores the address of the slab cache descriptor.

The mempool_create() function creates a new memory pool; it receives the number of memory elements min_nr, the addresses of the functions that implement the alloc and free methods, and an optional value for the pool_data field. The function allocates memory for the mempool_t object and the array of pointers to the memory elements, then repeatedly invokes the alloc method to get the min_nr memory elements. Conversely, the mempool_destroy() function releases all memory elements in the pool, then releases the array of elements and the mempool_t object themselves.

To allocate an element from a memory pool, the kernel invokes the mempool_alloc() function, passing to it the address of the mempool_t object and the memory allocation flags (see Table 8-5 and Table 8-6 earlier in this chapter). Essentially, the function tries to allocate a memory element from the underlying memory allocator by invoking the alloc method, according to the memory allocation flags specified as parameters. If the allocation succeeds, the function returns the memory element obtained, without touching the memory pool. Otherwise, if the allocation fails, the memory element is taken from the memory pool. Of course, too many allocations in a low-on-memory condition can exhaust the memory pool: in this case, if the __GFP_WAIT flag is not set, mempool_alloc() blocks the current process until a memory element is released to the memory pool.

Conversely, to release an element to a memory pool, the kernel invokes the mempool_free() function. If the memory pool is not full (curr_min is smaller than min_nr), the function adds the element to the memory pool. Otherwise, mempool_free() invokes the free method to release the element to the underlying memory allocator.

Noncontiguous Memory Area Management

We already know that it is preferable to map memory areas into sets of contiguous page frames, thus making better use of the cache and achieving lower average memory access times. Nevertheless, if the requests for memory areas are infrequent, it

makes sense to consider an allocation scheme based on noncontiguous page frames accessed through contiguous linear addresses. The main advantage of this schema is to avoid external fragmentation, while the disadvantage is that it is necessary to fiddle with the kernel Page Tables. Clearly, the size of a noncontiguous memory area must be a multiple of 4,096. Linux uses noncontiguous memory areas in several ways—for instance, to allocate data structures for active swap areas (see the section "Activating and Deactivating a Swap Area" in Chapter 17), to allocate space for a module (see Appendix B), or to allocate buffers to some I/O drivers. Furthermore, noncontiguous memory areas provide yet another way to make use of high memory page frames (see the later section "Allocating a Noncontiguous Memory Area").

Linear Addresses of Noncontiguous Memory Areas

To find a free range of linear addresses, we can look in the area starting from `PAGE_OFFSET` (usually `0xc0000000`, the beginning of the fourth gigabyte). Figure 8-7 shows how the fourth gigabyte linear addresses are used:

- The beginning of the area includes the linear addresses that map the first 896 MB of RAM (see the section "Process Page Tables" in Chapter 2); the linear address that corresponds to the end of the directly mapped physical memory is stored in the `high_memory` variable.
- The end of the area contains the fix-mapped linear addresses (see the section "Fix-Mapped Linear Addresses" in Chapter 2).
- Starting from `PKMAP_BASE` we find the linear addresses used for the persistent kernel mapping of high-memory page frames (see the section "Kernel Mappings of High-Memory Page Frames" earlier in this chapter).
- The remaining linear addresses can be used for noncontiguous memory areas. A safety interval of size 8 MB (macro `VMALLOC_OFFSET`) is inserted between the end of the physical memory mapping and the first memory area; its purpose is to "capture" out-of-bounds memory accesses. For the same reason, additional safety intervals of size 4 KB are inserted to separate noncontiguous memory areas.

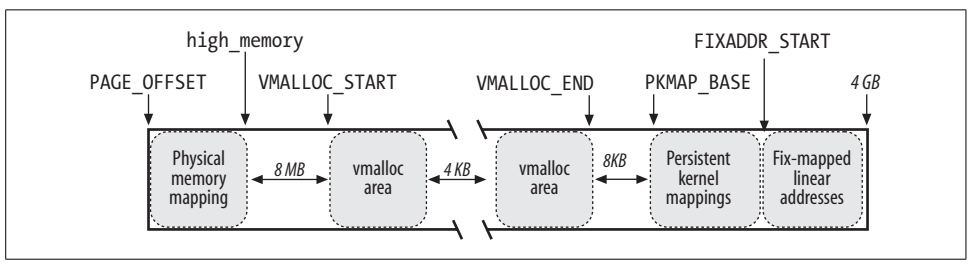


Figure 8-7. The linear address interval starting from PAGE_OFFSET

The VMALLOC_START macro defines the starting address of the linear space reserved for noncontiguous memory areas, while VMALLOC_END defines its ending address.

Descriptors of Noncontiguous Memory Areas

Each noncontiguous memory area is associated with a descriptor of type vm_struct, whose fields are listed in Table 8-13.

Table 8-13. The fields of the vm_struct descriptor

Type	Name	Description
void *	addr	Linear address of the first memory cell of the area
unsigned long	size	Size of the area plus 4,096 (inter-area safety interval)
unsigned long	flags	Type of memory mapped by the noncontiguous memory area
struct page **	pages	Pointer to array of nr_pages pointers to page descriptors
unsigned int	nr_pages	Number of pages filled by the area
unsigned long	phys_addr	Set to 0 unless the area has been created to map the I/O shared memory of a hardware device
struct vm_struct *	next	Pointer to next vm_struct structure

These descriptors are inserted in a simple list by means of the next field; the address of the first element of the list is stored in the vmlist variable. Accesses to this list are protected by means of the vmlist_lock read/write spin lock. The flags field identifies the type of memory mapped by the area: VM_ALLOC for pages obtained by means of vmalloc(), VM_MAP for already allocated pages mapped by means of vmap() (see the next section), and VM_IOREMAP for on-board memory of hardware devices mapped by means of ioremap() (see Chapter 13).

The get_vm_area() function looks for a free range of linear addresses between VMALLOC_START and VMALLOC_END. This function acts on two parameters: the size (size) in bytes of the memory region to be created, and a flag (flag) specifying the type of region (see above). The steps performed are the following:

1. Invokes kmalloc() to obtain a memory area for the new descriptor of type vm_struct.
2. Gets the vmlist_lock lock for writing and scans the list of descriptors of type vm_struct looking for a free range of linear addresses that includes at least size+4096 addresses (4096 is the size of the safety interval between the memory areas).
3. If such an interval exists, the function initializes the fields of the descriptor, releases the vmlist_lock lock, and terminates by returning the initial address of the noncontiguous memory area.
4. Otherwise, get_vm_area() releases the descriptor obtained previously, releases the vmlist_lock lock, and returns NULL.

Allocating a Noncontiguous Memory Area

The vmalloc( ) function allocates a noncontiguous memory area to the kernel. The parameter size denotes the size of the requested area. If the function is able to satisfy the request, it then returns the initial linear address of the new area; otherwise, it returns a NULL pointer:

```
void * vmalloc(unsigned long size)
{
    struct vm_struct *area;
    struct page **pages;
    unsigned int array_size, i;
    size = (size + PAGE_SIZE - 1) & PAGE_MASK;
    area = get_vm_area(size, VM_ALLOC);
    if (!area)
        return NULL;
    area->nr_pages = size >> PAGE_SHIFT;
    array_size = (area->nr_pages * sizeof(struct page *));
    area->pages = pages = kmalloc(array_size, GFP_KERNEL);
    if (!area_pages) {
        remove_vm_area(area->addr);
        kfree(area);
        return NULL;
    }
    memset(area->pages, 0, array_size);
    for (i=0; i<area->nr_pages; i++) {
        area->pages[i] = alloc_page(GFP_KERNEL|__GFP_HIGHMEM);
        if (!area->pages[i]) {
            area->nr_pages = i;
    fail:   vfree(area->addr);
            return NULL;
       }
    }
    if (map_vm_area(area, __pgprot(0x63), &pages))
        goto fail;
    return area->addr;
}
```

The function starts by rounding up the value of the size parameter to a multiple of 4,096 (the page frame size). Then vmalloc( ) invokes get_vm_area( ), which creates a new descriptor and returns the linear addresses assigned to the memory area. The flags field of the descriptor is initialized with the VM_ALLOC flag, which means that the noncontiguous page frames will be mapped into a linear address range by means of the vmalloc( ) function. Then the vmalloc( ) function invokes kmalloc( ) to request a group of contiguous page frames large enough to contain an array of page descriptor pointers. The memset( ) function is invoked to set all these pointers to NULL. Next the alloc_page( ) function is called repeatedly, once for each of the nr_pages of the region, to allocate a page frame and store the address of the corresponding page descriptor in the area->pages array. Observe that using the area->pages array is necessary because the page frames could belong to the ZONE_HIGHMEM memory zone, thus right now they are not necessarily mapped to a linear address.

Now comes the tricky part. Up to this point, a fresh interval of contiguous linear addresses has been obtained and a group of noncontiguous page frames has been allocated to map these linear addresses. The last crucial step consists of fiddling with the page table entries used by the kernel to indicate that each page frame allocated to the noncontiguous memory area is now associated with a linear address included in the interval of contiguous linear addresses yielded by `vmalloc()`. This is what `map_vm_area()` does.

The `map_vm_area()` function uses three parameters:

`area`
: The pointer to the `vm_struct` descriptor of the area.

`prot`
: The protection bits of the allocated page frames. It is always set to `0x63`, which corresponds to `Present`, `Accessed`, `Read/Write`, and `Dirty`.

`pages`
: The address of a variable pointing to an array of pointers to page descriptors (thus, `struct page ***` is used as the data type!).

The function starts by assigning the linear addresses of the start and end of the area to the `address` and `end` local variables, respectively:

```
address = area->addr;
end = address + (area->size - PAGE_SIZE);
```

Remember that `area->size` stores the actual size of the area plus the 4 KB inter-area safety interval. The function then uses the `pgd_offset_k` macro to derive the entry in the master kernel Page Global Directory related to the initial linear address of the area; it then acquires the kernel Page Table spin lock:

```
pgd = pgd_offset_k(address);
spin_lock(&init_mm.page_table_lock);
```

The function then executes the following cycle:

```
int ret = 0;
for (i = pgd_index(address); i < pgd_index(end-1); i++) {
    pud_t *pud = pud_alloc(&init_mm, pgd, address);
    ret = -ENOMEM;
    if (!pud)
        break;
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next < address || next > end)
        next = end;
    if (map_area_pud(pud, address, next, prot, pages))
        break;
    address = next;
    pgd++;
    ret = 0;
}
spin_unlock(&init_mm.page_table_lock);
```

```
flush_cache_vmap((unsigned long)area->addr, end);
return ret;
```

In each cycle, it first invokes `pud_alloc( )` to create a Page Upper Directory for the new area and writes its physical address in the right entry of the kernel Page Global Directory. It then calls `map_area_pud( )` to allocate all the page tables associated with the new Page Upper Directory. It adds the size of the range of linear addresses spanned by a single Page Upper Directory—the constant 2^{30} if PAE is enabled, 2^{22} otherwise—to the current value of `address`, and it increases the pointer `pgd` to the Page Global Directory.

The cycle is repeated until all Page Table entries referring to the noncontiguous memory area are set up.

The `map_area_pud( )` function executes a similar cycle for all the page tables that a Page Upper Directory points to:

```
do {
    pmd_t * pmd = pmd_alloc(&init_mm, pud, address);
    if (!pmd)
        return -ENOMEM;
    if (map_area_pmd(pmd, address, end-address, prot, pages))
        return -ENOMEM;
    address = (address + PUD_SIZE) & PUD_MASK;
    pud++;
} while (address < end);
```

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

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

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

The main cycle of `map_area_pte( )` is:

```
do {
    struct page * page = **pages;
    set_pte(pte, mk_pte(page, prot));
    address += PAGE_SIZE;
```

```
        pte++;
        (*pages)++;
    } while (address < end);
```

The page descriptor address page of the page frame to be mapped is read from the array's entry pointed to by the variable at address pages. The physical address of the new page frame is written into the Page Table by the set_pte and mk_pte macros. The cycle is repeated after adding the constant 4,096 (the length of a page frame) to address.

Notice that the Page Tables of the current process are not touched by map_vm_area(). Therefore, when a process in Kernel Mode accesses the noncontiguous memory area, a Page Fault occurs, because the entries in the process's Page Tables corresponding to the area are null. However, the Page Fault handler checks the faulty linear address against the master kernel Page Tables (which are init_mm.pgd Page Global Directory and its child page tables; see the section "Kernel Page Tables" in Chapter 2). Once the handler discovers that a master kernel Page Table includes a non-null entry for the address, it copies its value into the corresponding process's Page Table entry and resumes normal execution of the process. This mechanism is described in the section "Page Fault Exception Handler" in Chapter 9.

Beside the vmalloc() function, a noncontiguous memory area can be allocated by the vmalloc_32() function, which is very similar to vmalloc() but only allocates page frames from the ZONE_NORMAL and ZONE_DMA memory zones.

Linux 2.6 also features a vmap() function, which maps page frames already allocated in a noncontiguous memory area: essentially, this function receives as its parameter an array of pointers to page descriptors, invokes get_vm_area() to get a new vm_struct descriptor, and then invokes map_vm_area() to map the page frames. The function is thus similar to vmalloc(), but it does not allocate page frames.

Releasing a Noncontiguous Memory Area

The vfree() function releases noncontiguous memory areas created by vmalloc() or vmalloc_32(), while the vunmap() function releases memory areas created by vmap(). Both functions have one parameter—the address of the initial linear address of the area to be released; they both rely on the __vunmap() function to do the real work.

The __vunmap() function receives two parameters: the address addr of the initial linear address of the area to be released, and the flag deallocate_pages, which is set if the page frames mapped in the area should be released to the zoned page frame allocator (vfree()'s invocation), and cleared otherwise (vunmap()'s invocation). The function performs the following operations:

1. Invokes the remove_vm_area() function to get the address area of the vm_struct descriptor and to clear the kernel's page table entries corresponding to the linear address in the noncontiguous memory area.

2. If the deallocate_pages flag is set, it scans the area->pages array of pointers to the page descriptor; for each element of the array, invokes the __free_page( ) function to release the page frame to the zoned page frame allocator. Moreover, executes kfree(area->pages) to release the array itself.
3. Invokes kfree(area) to release the vm_struct descriptor.

The remove_vm_area( ) function performs the following cycle:

```
write_lock(&vmlist_lock);
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        unmap_vm_area(tmp);
        *p = tmp->next;
        break;
    }
}
write_unlock(&vmlist_lock);
return tmp;
```

The area itself is released by invoking unmap_vm_area( ). This function acts on a single parameter, namely a pointer area to the vm_struct descriptor of the area. It executes the following cycle to reverse the actions performed by map_vm_area( ):

```
address = area->addr;
end = address + area->size;
pgd = pgd_offset_k(address);
for (i = pgd_index(address); i <= pgd_index(end-1); i++) {
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next <= address || next > end)
        next = end;
    unmap_area_pud(pgd, address, next - address);
    address = next;
    pgd++;
}
```

In turn, unmap_area_pud( ) reverses the actions of map_area_pud( ) in the cycle:

```
do {
    unmap_area_pmd(pud, address, end-address);
    address = (address + PUD_SIZE) & PUD_MASK;
    pud++;
} while (address && (address < end));
```

The unmap_area_pmd( ) function reverses the actions of map_area_pmd( ) in the cycle:

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

Finally, unmap_area_pte( ) reverses the actions of map_area_pte( ) in the cycle:

```
do {
    pte_t page = ptep_get_and_clear(pte);
```

```
        address += PAGE_SIZE;
        pte++;
        if (!pte_none(page) && !pte_present(page))
            printk("Whee... Swapped out page in kernel page table\n");
    } while (address < end);
```

In every iteration of the cycle, the page table entry pointed to by `pte` is set to 0 by the `ptep_get_and_clear` macro.

As for `vmalloc()`, the kernel modifies the entries of the master kernel Page Global Directory and its child page tables (see the section "Kernel Page Tables" in Chapter 2), but it leaves unchanged the entries of the process page tables mapping the fourth gigabyte. This is fine because the kernel never reclaims Page Upper Directories, Page Middle Directories, and Page Tables rooted at the master kernel Page Global Directory.

For instance, suppose that a process in Kernel Mode accessed a noncontiguous memory area that later got released. The process's Page Global Directory entries are equal to the corresponding entries of the master kernel Page Global Directory, thanks to the mechanism explained in the section "Page Fault Exception Handler" in Chapter 9; they point to the same Page Upper Directories, Page Middle Directories, and Page Tables. The `unmap_area_pte()` function clears only the entries of the page tables (without reclaiming the page tables themselves). Further accesses of the process to the released noncontiguous memory area will trigger Page Faults because of the null page table entries. However, the handler will consider such accesses a bug, because the master kernel page tables do not include valid entries.

CHAPTER 10

System Calls

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

Unix systems implement most interfaces between User Mode processes and hardware devices by means of *system calls* issued to the kernel. This chapter examines in detail how Linux implements system calls that User Mode programs issue to the kernel.

POSIX APIs and System Calls

Let's start by stressing the difference between an application programmer interface (API) and a system call. The former is a function definition that specifies how to obtain a given service, while the latter is an explicit request to the kernel made via a software interrupt.

Unix systems include several libraries of functions that provide APIs to programmers. Some of the APIs defined by the *libc* standard C library refer to *wrapper routines* (routines whose only purpose is to issue a system call). Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ.

The converse is not true, by the way—an API does not necessarily correspond to a specific system call. First of all, the API could offer its services directly in User Mode. (For something abstract such as math functions, there may be no reason to make 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( )` APIs are implemented in the *libc* library. The code in this library keeps track of the allocation and deallocation requests and uses the `brk( )` system call to enlarge or shrink the process heap (see the section "Managing the Heap" in Chapter 9).

The POSIX standard refers to APIs and not to system calls. A system can be certified as POSIX-compliant if it offers the proper set of APIs to the application programs, no matter how the corresponding functions are implemented. As a matter of fact, several non-Unix systems have been certified as POSIX-compliant, because they offer all traditional Unix services in User Mode libraries.

From the programmer's point of view, the distinction between an API and a system call is irrelevant—the only things that matter are the function name, the parameter types, and the meaning of the return code. From the kernel designer's point of view, however, the distinction does matter because system calls belong to the kernel, while User Mode libraries don't.

Most wrapper routines return an integer value, whose meaning depends on the corresponding system call. A return value of –1 usually indicates that the kernel was unable to satisfy the process request. A failure in the system call handler may be caused by invalid parameters, a lack of available resources, hardware problems, and so on. The specific error code is contained in the `errno` variable, which is defined in the *libc* library.

Each error code is defined as a macro constant, which yields a corresponding positive integer value. The POSIX standard specifies the macro names of several error codes. In Linux, on 80×86 systems, these macros are defined in the header file *include/asm-i386/errno.h*. To allow portability of C programs among Unix systems, the *include/asm-i386/errno.h* header file is included, in turn, in the standard */usr/include/errno.h* C library header file. Other systems have their own specialized subdirectories of header files.

System Call Handler and Service Routines

When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function. As we will see in the next section, in the 80×86 architecture a Linux system call can be invoked in two different ways. The net result of both methods, however, is a jump to an assembly language function called the *system call handler*.

Because the kernel implements many different system calls, the User Mode process must pass a parameter called the *system call number* to identify the required system call; the `eax` register is used by Linux for this purpose. As we'll see in the section "Parameter Passing" later in this chapter, additional parameters are usually passed when invoking a system call.

All system calls return an integer value. The conventions for these return values are different from those for wrapper routines. In the kernel, positive or 0 values denote a successful termination of the system call, while negative values denote an error condition. In the latter case, the value is the negation of the error code that must be returned to the application program in the `errno` variable. The `errno` variable is not set or used by the kernel. Instead, the wrapper routines handle the task of setting this variable after a return from a system call.

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

- Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language).
- Handles the system call by invoking a corresponding C function called the *system call service routine*.
- Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language).

The name of the service routine associated with the *xyz*( ) system call is usually `sys_`*xyz*( ); there are, however, a few exceptions to this rule.

Figure 10-1 illustrates the relationships between the application program that invokes a system call, the corresponding wrapper routine, the system call handler, and the system call service routine. The arrows denote the execution flow between the functions. The terms "`SYSCALL`" and "`SYSEXIT`" are placeholders for the actual assembly language instructions that switch the CPU, respectively, from User Mode to Kernel Mode and from Kernel Mode to User Mode.

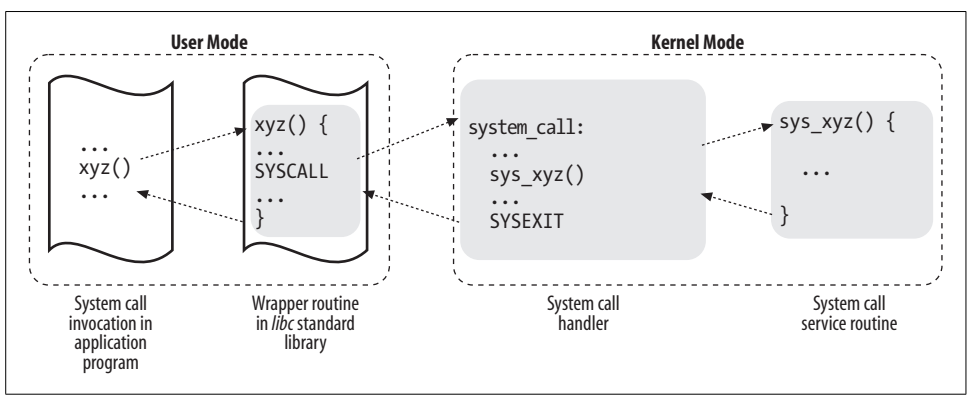


Figure 10-1. Invoking a system call

To associate each system call number with its corresponding service routine, the kernel uses a *system call dispatch table*, which is stored in the `sys_call_table` array and

has NR_syscalls entries (289 in the Linux 2.6.11 kernel). The 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, each entry of the dispatch table may contain the address of the sys_ni_syscall( ) function, which is the service routine of the "nonimplemented" system calls; it just returns the error code -ENOSYS.

Entering and Exiting a System Call

Native applications* can invoke a system call in two different ways:

- By executing the int $0x80 assembly language instruction; in older versions of the Linux kernel, this was the only way to switch from User Mode to Kernel Mode.
- By executing the sysenter assembly language instruction, introduced in the Intel Pentium II microprocessors; this instruction is now supported by the Linux 2.6 kernel.

Similarly, the kernel can exit from a system call—thus switching the CPU back to User Mode—in two ways:

- By executing the iret assembly language instruction.
- By executing the sysexit assembly language instruction, which was introduced in the Intel Pentium II microprocessors together with the sysenter instruction.

However, supporting two different ways to enter the kernel is not as simple as it might look, because:

- The kernel must support both older libraries that only use the int $0x80 instruction and more recent ones that also use the sysenter instruction.
- A standard library that makes use of the sysenter instruction must be able to cope with older kernels that support only the int $0x80 instruction.
- The kernel and the standard library must be able to run both on older processors that do not include the sysenter instruction and on more recent ones that include it.

We will see in the section "Issuing a System Call via the sysenter Instruction" later in this chapter how the Linux kernel solves these compatibility problems.

* As we will see in the section "Execution Domains" in Chapter 20, Linux can execute programs compiled for "foreign" operating systems. Therefore, the kernel offers a compatibility mode to enter a system call: User Mode processes executing iBCS and Solaris/x86 programs can enter the kernel by jumping into suitable call gates included in the default Local Descriptor Table (see the section "The Linux LDTs" in Chapter 2).

Issuing a System Call via the int $0x80 Instruction

The "traditional" way to invoke a system call makes use of the `int` assembly language instruction, which was discussed in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4.

The vector 128—in hexadecimal, `0x80`—is associated with the kernel entry point. The `trap_init()` function, invoked during kernel initialization, sets up the Interrupt Descriptor Table entry corresponding to vector 128 as follows:

```
set_system_gate(0x80, &system_call);
```

The call loads the following values into the gate descriptor fields (see the section "Interrupt, Trap, and System Gates" in Chapter 4):

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

Offset
: The pointer to the `system_call()` system call handler.

Type
: Set to 15. Indicates that the exception is a Trap and that the corresponding handler does not disable maskable interrupts.

DPL (Descriptor Privilege Level)
: Set to 3. This allows processes in User Mode to invoke the exception handler (see the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4).

Therefore, when a User Mode process issues an `int $0x80` instruction, the CPU switches into Kernel Mode and starts executing instructions from the `system_call` address.

The system_call() function

The `system_call()` function starts by saving the system call number and all the CPU registers that may be used by the exception handler on the stack—except for `eflags`, `cs`, `eip`, `ss`, and `esp`, which have already been saved automatically by the control unit (see the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4). The `SAVE_ALL` macro, which was already discussed in the section "I/O Interrupt Handling" in Chapter 4, also loads the Segment Selector of the kernel data segment in `ds` and `es`:

```
system_call:
  pushl %eax
  SAVE_ALL
  movl $0xfffffe000, %ebx /* or 0xffffe000 for 4-KB stacks */
  andl %esp, %ebx
```

The function then stores the address of the `thread_info` data structure of the current process in `ebx` (see the section "Identifying a Process" in Chapter 3). This is done by

taking the value of the kernel stack pointer and rounding it up to a multiple of 4 or 8 KB (see the section "Identifying a Process" in Chapter 3).

Next, the system_call( ) function checks whether either one of the TIF_SYSCALL_TRACE and TIF_SYSCALL_AUDIT flags included in the flags field of the thread_info structure is set—that is, whether the system call invocations of the executed program are being traced by a debugger. If this is the case, system_call( ) invokes the do_syscall_trace( ) function twice: once right before and once right after the execution of the system call service routine (as described later). This function stops current and thus allows the debugging process to collect information about it.

A validity check is then performed on the system call number passed by the User Mode process. If it is greater than or equal to the number of entries in the system call dispatch table, the system call handler terminates:

```
    cmpl $NR_syscalls, %eax
    jb nobadsys
    movl $(-ENOSYS), 24(%esp)
    jmp resume_userspace
nobadsys:
```

If the system call number is not valid, the function stores the -ENOSYS value in the stack location where the eax register has been saved—that is, at offset 24 from the current stack top. It then jumps to resume_userspace (see below). In this way, when the process resumes its execution in User Mode, it will find a negative return code in eax.

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

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

Because each entry in the dispatch table is 4 bytes long, the kernel finds the address of the service routine to be invoked by multiplying the system call number by 4, adding the initial address of the sys_call_table dispatch table, and extracting a pointer to the service routine from that slot in the table.

Exiting from the system call

When the system call service routine terminates, the system_call( ) function gets its return code from eax and stores it in the stack location where the User Mode value of the eax register is saved:

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

Thus, the User Mode process will find the return code of the system call in the eax register.

Then, the system_call( ) function disables the local interrupts and checks the flags in the thread_info structure of current:

```
    cli
    movl 8(%ebp), %ecx
```

```
testw $0xffff, %cx
je restore_all
```

The flags field is at offset 8 in the thread_info structure; the mask 0xffff selects the bits corresponding to all flags listed in Table 4-15 except TIF_POLLING_NRFLAG. If none of these flags is set, the function jumps to the restore_all label: as described in the section "Returning from Interrupts and Exceptions" in Chapter 4, this code restores the contents of the registers saved on the Kernel Mode stack and executes an iret assembly language instruction to resume the User Mode process. (You might refer to the flow diagram in Figure 4-6.)

If any of the flags is set, then there is some work to be done before returning to User Mode. If the TIF_SYSCALL_TRACE flag is set, the system_call() function invokes for the second time the do_syscall_trace() function, then jumps to the resume_userspace label. Otherwise, if the TIF_SYSCALL_TRACE flag is not set, the function jumps to the work_pending label.

As explained in the section "Returning from Interrupts and Exceptions" in Chapter 4, that code at the resume_userspace and work_pending labels checks for rescheduling requests, virtual-8086 mode, pending signals, and single stepping; then eventually a jump is done to the restore_all label to resume the execution of the User Mode process.

Issuing a System Call via the sysenter Instruction

The int assembly language instruction is inherently slow because it performs several consistency and security checks. (The instruction is described in detail in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4.)

The sysenter instruction, dubbed in Intel documentation as "Fast System Call," provides a faster way to switch from User Mode to Kernel Mode.

The sysenter instruction

The sysenter assembly language instruction makes use of three special registers that must be loaded with the following information:*

SYSENTER_CS_MSR
: The Segment Selector of the kernel code segment

SYSENTER_EIP_MSR
: The linear address of the kernel entry point

SYSENTER_ESP_MSR
: The kernel stack pointer

* "MSR" is an acronym for "Model-Specific Register" and denotes a register that is present only in some models of 80×86 microprocessors.

When the sysenter instruction is executed, the CPU control unit:

1. Copies the content of SYSENTER_CS_MSR into cs.
2. Copies the content of SYSENTER_EIP_MSR into eip.
3. Copies the content of SYSENTER_ESP_MSR into esp.
4. Adds 8 to the value of SYSENTER_CS_MSR, and loads this value into ss.

Therefore, the CPU switches to Kernel Mode and starts executing the first instruction of the kernel entry point. As we have seen in the section "The Linux GDT" in Chapter 2, the kernel stack segment coincides with the kernel data segment, and the corresponding descriptor follows the descriptor of the kernel code segment in the Global Descriptor Table; therefore, step 4 loads the proper Segment Selector in the ss register.

The three model-specific registers are initialized by the enable_sep_cpu() function, which is executed once by every CPU in the system during the initialization of the kernel. The function performs the following steps:

1. Writes the Segment Selector of the kernel code (__KERNEL_CS) in the SYSENTER_CS_MSR register.
2. Writes in the SYSENTER_CS_EIP register the linear address of the sysenter_entry() function described below.
3. Computes the linear address of the end of the local TSS, and writes this value in the SYSENTER_CS_ESP register.*

The setting of the SYSENTER_CS_ESP register deserves some comments. When a system call starts, the kernel stack is empty, thus the esp register should point to the end of the 4- or 8-KB memory area that includes the kernel stack and the descriptor of the current process (see Figure 3-2). The User Mode wrapper routine cannot properly set this register, because it does not know the address of this memory area; on the other hand, the value of the register must be set before switching to Kernel Mode. Therefore, the kernel initializes the register so as to encode the address of the Task State Segment of the local CPU. As we have described in step 3 of the __switch_to() function (see the section "Performing the Process Switch" in Chapter 3), at every process switch the kernel saves the kernel stack pointer of the current process in the esp0 field of the local TSS. Thus, the system call handler reads the esp register, computes the address of the esp0 field of the local TSS, and loads into the same esp register the proper kernel stack pointer.

* The encoding of the local TSS address written in SYSENTER_ESP_MSR is due to the fact that the register should point to a real stack, which grows towards lower address. In practice, initializing the register with any value would work, provided that it is possible to get the address of the local TSS from such a value.

The vsyscall page

A wrapper function in the *libc* standard library can make use of the sysenter instruction only if both the CPU and the Linux kernel support it.

This compatibility problem calls for a quite sophisticated solution. Essentially, in the initialization phase the sysenter_setup() function builds a page frame called *vsyscall page* containing a small ELF shared object (i.e., a tiny ELF dynamic library). When a process issues an execve() system call to start executing an ELF program, the code in the vsyscall page is dynamically linked to the process address space (see the section "The exec Functions" in Chapter 20). The code in the vsyscall page makes use of the best available instruction to issue a system call.

The sysenter_setup() function allocates a new page frame for the vsyscall page and associates its physical address with the FIX_VSYSCALL fix-mapped linear address (see the section "Fix-Mapped Linear Addresses" in Chapter 2). Then, the function copies in the page either one of two predefined ELF shared objects:

- If the CPU does not support sysenter, the function builds a vsyscall page that includes the code:

```
__kernel_vsyscall:
  int $0x80
  ret
```

- Otherwise, if the CPU does support sysenter, the function builds a vsyscall page that includes the code:

```
__kernel_vsyscall:
  pushl %ecx
  pushl %edx
  pushl %ebp
  movl %esp, %ebp
  sysenter
```

When a wrapper routine in the standard library must invoke a system call, it calls the __kernel_vsyscall() function, whatever it may be.

A final compatibility problem is due to old versions of the Linux kernel that do not support the sysenter instruction; in this case, of course, the kernel does not build the vsyscall page and the __kernel_vsyscall() function is not linked to the address space of the User Mode processes. When recent standard libraries recognize this fact, they simply execute the int $0x80 instruction to invoke the system calls.

Entering the system call

The sequence of steps performed when a system call is issued via the sysenter instruction is the following:

1. The wrapper routine in the standard library loads the system call number into the eax register and calls the __kernel_vsyscall() function.

2. The __kernel_vsyscall() function saves on the User Mode stack the contents of ebp, edx, and ecx (these registers are going to be used by the system call handler), copies the user stack pointer in ebp, then executes the sysenter instruction.
3. The CPU switches from User Mode to Kernel Mode, and the kernel starts executing the sysenter_entry() function (pointed to by the SYSENTER_EIP_MSR register).
4. The sysenter_entry() assembly language function performs the following steps:
   a. Sets up the kernel stack pointer:

      ```
      movl -508(%esp), %esp
      ```

      Initially, the esp register points to the first location after the local TSS, which is 512bytes long. Therefore, the instruction loads in the esp register the contents of the field at offset 4 in the local TSS, that is, the contents of the esp0 field. As already explained, the esp0 field always stores the kernel stack pointer of the current process.
   b. Enables local interrupts:

      ```
      sti
      ```

   c. Saves in the Kernel Mode stack the Segment Selector of the user data segment, the current user stack pointer, the eflags register, the Segment Selector of the user code segment, and the address of the instruction to be executed when exiting from the system call:

      ```
      pushl $(__USER_DS)
      pushl %ebp
      pushfl
      pushl $(__USER_CS)
      pushl $SYSENTER_RETURN
      ```

      Observe that these instructions emulate some operations performed by the int assembly language instruction (steps 5c and 7 in the description of int in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4).
   d. Restores in ebp the original value of the register passed by the wrapper routine:

      ```
      movl (%ebp), %ebp
      ```

      This instruction does the job, because __kernel_vsyscall() saved on the User Mode stack the original value of ebp and then loaded in ebp the current value of the user stack pointer.
   e. Invokes the system call handler by executing a sequence of instructions identical to that starting at the system_call label described in the earlier section "Issuing a System Call via the int $0x80 Instruction."

Exiting from the system call

When the system call service routine terminates, the sysenter_entry() function executes essentially the same operations as the system_call() function (see previous

section). First, it gets the return code of the system call service routine from eax and stores it in the kernel stack location where the User Mode value of the eax register is saved. Then, the function disables the local interrupts and checks the flags in the thread_info structure of current.

If any of the flags is set, then there is some work to be done before returning to User Mode. In order to avoid code duplication, this case is handled exactly as in the system_call() function, thus the function jumps to the resume_userspace or work_pending labels (see flow diagram in Figure 4-6 in Chapter 4). Eventually, the iret assembly language instruction fetches from the Kernel Mode stack the five arguments saved in step 4c by the sysenter_entry() function, and thus switches the CPU back to User Mode and starts executing the code at the SYSENTER_RETURN label (see below).

If the sysenter_entry() function determines that the flags are cleared, it performs a quick return to User Mode:

```
movl 40(%esp), %edx
movl 52(%esp), %ecx
xorl %ebp, %ebp
sti
sysexit
```

The edx and ecx registers are loaded with a couple of the stack values saved by sysenter_entry() in step 4c in the previos section: edx gets the address of the SYSENTER_RETURN label, while ecx gets the current user data stack pointer.

The sysexit instruction

The sysexit assembly language instruction is the companion of sysenter: it allows a fast switch from Kernel Mode to User Mode. When the instruction is executed, the CPU control unit performs the following steps:

1. Adds 16 to the value in the SYSENTER_CS_MSR register, and loads the result in the cs register.
2. Copies the content of the edx register into the eip register.
3. Adds 24 to the value in the SYSENTER_CS_MSR register, and loads the result in the ss register.
4. Copies the content of the ecx register into the esp register.

Because the SYSENTER_CS_MSR register is loaded with the Segment Selector of the kernel code, the cs register is loaded with the Segment Selector of the user code, while the ss register is loaded with the Segment Selector of the user data segment (see the section "The Linux GDT" in Chapter 2).

As a result, the CPU switches from Kernel Mode to User Mode and starts executing the instruction whose address is stored in the edx register.

The SYSENTER_RETURN code

The code at the SYSENTER_RETURN label is stored in the vsyscall page, and it is executed when a system call entered via sysenter is being terminated, either by the iret instruction or the sysexit instruction.

The code simply restores the original contents of the ebp, edx, and ecx registers saved in the User Mode stack, and returns the control to the wrapper routine in the standard library:

```
SYSENTER_RETURN:
  popl %ebp
  popl %edx
  popl %ecx
  ret
```

Parameter Passing

Like ordinary functions, system calls often require some input/output parameters, which may consist of actual values (i.e., numbers), addresses of variables in the address space of the User Mode process, or even addresses of data structures including pointers to User Mode functions (see the section "System Calls Related to Signal Handling" in Chapter 11).

Because the system_call() and the sysenter_entry() functions are the common entry points for all system calls in Linux, each of them has at least one parameter: the system call number passed in the eax register. For instance, if an application program invokes the fork() wrapper routine, the eax register is set to 2 (i.e., __NR_fork) before executing the int $0x80 or sysenter assembly language instruction. Because the register is set by the wrapper routines included in the *libc* library, programmers do not usually care about the system call number.

The fork() system call does not require other parameters. However, many system calls do require additional parameters, which must be explicitly passed by the application program. For instance, the mmap() system call may require up to six additional parameters (besides the system call number).

The parameters of ordinary C functions are usually passed by writing their values in the active program stack (either the User Mode stack or the Kernel Mode stack). Because system calls are a special kind of function that cross over from user to kernel land, neither the User Mode or the Kernel Mode stacks can be used. Rather, system call parameters are written in the CPU registers before issuing the system call. The kernel then copies the parameters stored in the CPU registers onto the Kernel Mode stack before invoking the system call service routine, because the latter is an ordinary C function.

Why doesn't the kernel copy parameters directly from the User Mode stack to the Kernel Mode stack? First of all, working with two stacks at the same time is complex;

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

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

- The length of each parameter cannot exceed the length of a register (32 bits).*
- The number of parameters must not exceed six, besides the system call number passed in eax, because 80×86 processors have a very limited number of registers.

The first condition is always true because, according to the POSIX standard, large parameters that cannot be stored in a 32-bit register must be passed by reference. A typical example is the settimeofday( ) system call, which must read a 64-bit structure.

However, system calls that require more than six parameters exist. In such cases, a single register is used to point to a memory area in the process address space that contains the parameter values. Of course, programmers do not have to care about this workaround. As with every C function call, parameters are automatically saved on the stack when the wrapper routine is invoked. This routine will find the appropriate way to pass the parameters to the kernel.

The registers used to store the system call number and its parameters are, in increasing order, eax (for the system call number), ebx, ecx, edx, esi, edi, and ebp. As seen before, system_call( ) and sysenter_entry( ) save the values of these registers on the Kernel Mode stack by using the SAVE_ALL macro. Therefore, when the system call service routine goes to the stack, it finds the return address to system_call( ) or to sysenter_entry( ), followed by the parameter stored in ebx (the first parameter of the system call), the parameter stored in ecx, and so on (see the section "Saving the registers for the interrupt handler" in Chapter 4). This stack configuration is exactly the same as in an ordinary function call, and therefore the service routine can easily refer to its parameters by using the usual C-language constructs.

Let's look at an example. The sys_write( ) service routine, which handles the write( ) system call, is declared as:

```
int sys_write (unsigned int fd, const char * buf, unsigned int count)
```

The C compiler produces an assembly language function that expects to find the fd, buf, and count parameters on top of the stack, right below the return address, in the locations used to save the contents of the ebx, ecx, and edx registers, respectively.

In a few cases, even if the system call doesn't use any parameters, the corresponding service routine needs to know the contents of the CPU registers right before the system call was issued. For example, the do_fork( ) function that implements fork( ) needs to know the value of the registers in order to duplicate them in the child process thread field (see the section "The thread field" in Chapter 3). In these cases, a

* We refer, as usual, to the 32-bit architecture of the 80×86 processors. The discussion in this section does not apply to 64-bit architectures.

single parameter of type pt_regs allows the service routine to access the values saved in the Kernel Mode stack by the SAVE_ALL macro (see the section "The do_IRQ() function" in Chapter 4):

```
int sys_fork (struct pt_regs regs)
```

The return value of a service routine must be written into the eax register. This is automatically done by the C compiler when a `return n;` instruction is executed.

Verifying the Parameters

All system call parameters must be carefully checked before the kernel attempts to satisfy a user request. The type of check depends both on the system call and on the specific parameter. Let's go back to the write() system call introduced before: the fd parameter should be a file descriptor that identifies a specific file, so sys_write() must check whether fd really is a file descriptor of a file previously opened and whether the process is allowed to write into it (see the section "File-Handling System Calls" in Chapter 1). If any of these conditions are not true, the handler must return a negative value—in this case, the error code -EBADF.

One type of checking, however, is common to all system calls. Whenever a parameter specifies an address, the kernel must check whether it is inside the process address space. There are two possible ways to perform this check:

- Verify that the linear address belongs to the process address space and, if so, that the memory region including it has the proper access rights.
- Verify just that the linear address is lower than PAGE_OFFSET (i.e., that it doesn't fall within the range of interval addresses reserved to the kernel).

Early Linux kernels performed the first type of checking. But it is quite time consuming because it must be executed for each address parameter included in a system call; furthermore, it is usually pointless because faulty programs are not very common.

Therefore, starting with Version 2.2, Linux employs the second type of checking. This is much more efficient because it does not require any scan of the process memory region descriptors. Obviously, this is a very coarse check: verifying that the linear address is smaller than PAGE_OFFSET is a necessary but not sufficient condition for its validity. But there's no risk in confining the kernel to this limited kind of check because other errors will be caught later.

The approach followed is thus to defer the real checking until the last possible moment—that is, until the Paging Unit translates the linear address into a physical one. We will discuss in the section "Dynamic Address Checking: The Fix-up Code," later in this chapter, how the Page Fault exception handler succeeds in detecting those bad addresses issued in Kernel Mode that were passed as parameters by User Mode processes.

One might wonder at this point why the coarse check is performed at all. This type of checking is actually crucial to preserve both process address spaces and the kernel address space from illegal accesses. We saw in Chapter 2 that the RAM is mapped starting from PAGE_OFFSET. This means that kernel routines are able to address all pages present in memory. Thus, if the coarse check were not performed, a User Mode process might pass an address belonging to the kernel address space as a parameter and then be able to read or write every page present in memory without causing a Page Fault exception.

The check on addresses passed to system calls is performed by the access_ok() macro, which acts on two parameters: addr and size. The macro checks the address interval delimited by addr and addr + size – 1. It is essentially equivalent to the following C function:

```
int access_ok(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a ||
        a + size > current_thread_info()->addr_limit.seg)
        return 0;
    return 1;
}
```

The function first verifies whether addr + size, the highest address to be checked, is larger than $2^{32}-1$; because unsigned long integers and pointers are represented by the GNU C compiler (gcc) as 32-bit numbers, this is equivalent to checking for an overflow condition. The function also checks whether addr + size exceeds the value stored in the addr_limit.seg field of the thread_info structure of current. This field usually has the value PAGE_OFFSET for normal processes and the value 0xffffffff for kernel threads. The value of the addr_limit.seg field can be dynamically changed by the get_fs and set_fs macros; this allows the kernel to bypass the security checks made by access_ok(), so that it can invoke system call service routines, directly passing to them addresses in the kernel data segment.

The verify_area() function performs the same check as the access_ok() macro; although this function is considered obsolete, it is still widely used in the source code.

Accessing the Process Address Space

System call service routines often need to read or write data contained in the process's address space. Linux includes a set of macros that make this access easier. We'll describe two of them, called get_user() and put_user(). The first can be used to read 1, 2, or 4 consecutive bytes from an address, while the second can be used to write data of those sizes into an address.

Each function accepts two arguments, a value x to transfer and a variable ptr. The second variable also determines how many bytes to transfer. Thus, in get_user(x,ptr),

the size of the variable pointed to by ptr causes the function to expand into a __get_user_1( ), __get_user_2( ), or __get_user_4( ) assembly language function. Let's consider one of them, __get_user_2( ):

```
__get_user_2:
    addl $1, %eax
    jc bad_get_user
    movl $0xfffffe000, %edx /* or 0xfffff000 for 4-KB stacks */
    andl %esp, %edx
    cmpl 24(%edx), %eax
    jae bad_get_user
2:  movzwl -1(%eax), %edx
    xorl %eax, %eax
    ret
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
```

The eax register contains the address ptr of the first byte to be read. The first six instructions essentially perform the same checks as the access_ok( ) macro: they ensure that the 2 bytes to be read have addresses less than 4 GB as well as less than the addr_limit.seg field of the current process. (This field is stored at offset 24 in the thread_info structure of current, which appears in the first operand of the cmpl instruction.)

If the addresses are valid, the function executes the movzwl instruction to store the data to be read in the two least significant bytes of edx register while setting the high-order bytes of edx to 0; then it sets a 0 return code in eax and terminates. If the addresses are not valid, the function clears edx, sets the -EFAULT value into eax, and terminates.

The put_user(x,ptr) macro is similar to the one discussed before, except it writes the value x into the process address space starting from address ptr. Depending on the size of x, it invokes either the __put_user_asm( ) macro (size of 1, 2, or 4 bytes) or the __put_user_u64( ) macro (size of 8 bytes). Both macros return the value 0 in the eax register if they succeed in writing the value, and -EFAULT otherwise.

Several other functions and macros are available to access the process address space in Kernel Mode; they are listed in Table 10-1. Notice that many of them also have a variant prefixed by two underscores (__). The ones without initial underscores take extra time to check the validity of the linear address interval requested, while the ones with the underscores bypass that check. Whenever the kernel must repeatedly access the same memory area in the process address space, it is more efficient to check the address once at the start and then access the process area without making any further checks.

Table 10-1. Functions and macros that access the process address space

Function	Action
`get_user __get_user`	Reads an integer value from user space (1, 2, or 4 bytes)
`put_user __put_user`	Writes an integer value to user space (1, 2, or 4 bytes)
`copy_from_user __copy_from_user`	Copies a block of arbitrary size from user space
`copy_to_user __copy_to_user`	Copies a block of arbitrary size to user space
`strncpy_from_user __strncpy_from_user`	Copies a null-terminated string from user space
`strlen_user strnlen_user`	Returns the length of a null-terminated string in user space
`clear_user __clear_user`	Fills a memory area in user space with zeros

Dynamic Address Checking: The Fix-up Code

As seen previously, `access_ok()` makes a coarse check on the validity of linear addresses passed as parameters of a system call. This check only ensures that the User Mode process is not attempting to fiddle with the kernel address space; however, the linear addresses passed as parameters still might not belong to the process address space. In this case, a Page Fault exception will occur when the kernel tries to use any of such bad addresses.

Before describing how the kernel detects this type of error, let's specify the three cases in which Page Fault exceptions may occur in Kernel Mode. These cases must be distinguished by the Page Fault handler, because the actions to be taken are quite different.

1. The kernel attempts to address a page belonging to the process address space, but either the corresponding page frame does not exist or the kernel tries to write a read-only page. In these cases, the handler must allocate and initialize a new page frame (see the sections "Demand Paging" and "Copy On Write" in Chapter 9).
2. The kernel addresses a page belonging to its address space, but the corresponding Page Table entry has not yet been initialized (see the section "Handling Noncontiguous Memory Area Accesses" in Chapter 9). In this case, the kernel must properly set up some entries in the Page Tables of the current process.
3. Some kernel functions include a programming bug that causes the exception to be raised when that program is executed; alternatively, the exception might be caused by a transient hardware error. When this occurs, the handler must perform a kernel oops (see the section "Handling a Faulty Address Inside the Address Space" in Chapter 9).
4. The case introduced in this chapter: a system call service routine attempts to read or write into a memory area whose address has been passed as a system call parameter, but that address does not belong to the process address space.

The Page Fault handler can easily recognize the first case by determining whether the faulty linear address is included in one of the memory regions owned by the process.

It is also able to detect the second case by checking whether the corresponding master kernel Page Table entry includes a proper non-null entry that maps the address. Let's now explain how the handler distinguishes the remaining two cases.

The Exception Tables

The key to determining the source of a Page Fault lies in the narrow range of calls that the kernel uses to access the process address space. Only the small group of functions and macros described in the previous section are used to access this address space; thus, if the exception is caused by an invalid parameter, the instruction that caused it *must* be included in one of the functions or else be generated by expanding one of the macros. The number of the instructions that address user space is fairly small.

Therefore, it does not take much effort to put the address of each kernel instruction that accesses the process address space into a structure called the *exception table*. If we succeed in doing this, the rest is easy. When a Page Fault exception occurs in Kernel Mode, the `do_page_fault()` handler examines the exception table: if it includes the address of the instruction that triggered the exception, the error is caused by a bad system call parameter; otherwise, it is caused by a more serious bug.

Linux defines several exception tables. The main exception table is automatically generated by the C compiler when building the kernel program image. It is stored in the `__ex_table` section of the kernel code segment, and its starting and ending addresses are identified by two symbols produced by the C compiler: `__start___ex_table` and `__stop___ex_table`.

Moreover, each dynamically loaded module of the kernel (see Appendix B) includes its own local exception table. This table is automatically generated by the C compiler when building the module image, and it is loaded into memory when the module is inserted in the running kernel.

Each entry of an exception table is an `exception_table_entry` structure that has two fields:

`insn`
: The linear address of an instruction that accesses the process address space

`fixup`
: The address of the assembly language code to be invoked when a Page Fault exception triggered by the instruction located at `insn` occurs

The fixup code consists of a few assembly language instructions that solve the problem triggered by the exception. As we will see later in this section, the fix usually consists of inserting a sequence of instructions that forces the service routine to return an error code to the User Mode process. These instructions, which are usually defined in the same macro or function that accesses the process address space, are placed by the C compiler into a separate section of the kernel code segment called `.fixup`.

The search_exception_tables( ) function is used to search for a specified address in all exception tables: if the address is included in a table, the function returns a pointer to the corresponding exception_table_entry structure; otherwise, it returns NULL. Thus the Page Fault handler do_page_fault( ) executes the following statements:

```
if ((fixup = search_exception_tables(regs->eip))) {
    regs->eip = fixup->fixup;
    return 1;
}
```

The regs->eip field contains the value of the eip register saved on the Kernel Mode stack when the exception occurred. If the value in the register (the instruction pointer) is in an exception table, do_page_fault( ) replaces the saved value with the address found in the entry returned by search_exception_tables( ). Then the Page Fault handler terminates and the interrupted program resumes with execution of the fixup code.

Generating the Exception Tables and the Fixup Code

The GNU Assembler .section directive allows programmers to specify which section of the executable file contains the code that follows. As we will see in Chapter 20, an executable file includes a code segment, which in turn may be subdivided into sections. Thus, the following assembly language instructions add an entry into an exception table; the "a" attribute specifies that the section must be loaded into memory together with the rest of the kernel image:

```
.section __ex_table, "a"
    .long faulty_instruction_address, fixup_code_address
.previous
```

The .previous directive forces the assembler to insert the code that follows into the section that was active when the last .section directive was encountered.

Let's consider again the __get_user_1( ), __get_user_2( ), and __get_user_4( ) functions mentioned before. The instructions that access the process address space are those labeled as 1, 2, and 3:

```
__get_user_1:
    [...]
1:  movzbl (%eax), %edx
    [...]
__get_user_2:
    [...]
2:  movzwl -1(%eax), %edx
    [...]
__get_user_4:
    [...]
3:  movl -3(%eax), %edx
    [...]
bad_get_user:
    xorl %edx, %edx
```

```
        movl $-EFAULT, %eax
        ret
    .section __ex_table,"a"
        .long 1b, bad_get_user
        .long 2b, bad_get_user
        .long 3b, bad_get_user
    .previous
```

Each exception table entry consists of two labels. The first one is a numeric label with a `b` suffix to indicate that the label is "backward;" in other words, it appears in a previous line of the program. The fixup code is common to the three functions and is labeled as `bad_get_user`. If a Page Fault exception is generated by the instructions at label `1`, `2`, or `3`, the fixup code is executed. It simply returns an `-EFAULT` error code to the process that issued the system call.

Other kernel functions that act in the User Mode address space use the fixup code technique. Consider, for instance, the `strlen_user(string)` macro. This macro returns either the length of a null-terminated string passed as a parameter in a system call or the value 0 on error. The macro essentially yields the following assembly language instructions:

```
        movl $0, %eax
        movl $0x7fffffff, %ecx
        movl %ecx, %ebx
        movl string, %edi
    0:  repne; scasb
        subl %ecx, %ebx
        movl %ebx, %eax
    1:
    .section .fixup,"ax"
    2:  xorl %eax, %eax
        jmp 1b
    .previous
    .section __ex_table,"a"
        .long 0b, 2b
    .previous
```

The `ecx` and `ebx` registers are initialized with the `0x7fffffff` value, which represents the maximum allowed length for the string in the User Mode address space. The `repne;scasb` assembly language instructions iteratively scan the string pointed to by the `edi` register, looking for the value 0 (the end of string `\0` character) in `eax`. Because `scasb` decreases the `ecx` register at each iteration, the `eax` register ultimately stores the total number of bytes scanned in the string (that is, the length of the string).

The fixup code of the macro is inserted into the `.fixup` section. The `"ax"` attributes specify that the section must be loaded into memory and that it contains executable code. If a Page Fault exception is generated by the instructions at label `0`, the fixup code is executed; it simply loads the value 0 in `eax`—thus forcing the macro to return a 0 error code instead of the string length—and then jumps to the `1` label, which corresponds to the instruction following the macro.

The second `.section` directive adds an entry containing the address of the `repne; scasb` instruction and the address of the corresponding fixup code in the `__ex_table` section.

Kernel Wrapper Routines

Although system calls are used mainly by User Mode processes, they can also be invoked by kernel threads, which cannot use library functions. To simplify the declarations of the corresponding wrapper routines, Linux defines a set of seven macros called `_syscall0` through `_syscall6`.

In the name of each macro, the numbers 0 through 6 correspond to the number of parameters used by the system call (excluding the system call number). The macros are used to declare wrapper routines that are not already included in the *libc* standard library (for instance, because the Linux system call is not yet supported by the library); however, they cannot be used to define wrapper routines for system calls that have more than six parameters (excluding the system call number) or for system calls that yield nonstandard return values.

Each macro requires exactly $2 + 2 \times n$ parameters, with n being the number of parameters of the system call. The first two parameters specify the return type and the name of the system call; each additional pair of parameters specifies the type and the name of the corresponding system call parameter. Thus, for instance, the wrapper routine of the `fork( )` system call may be generated by:

```
_syscall0(int,fork)
```

while the wrapper routine of the `write( )` system call may be generated by:

```
_syscall3(int,write,int,fd,const char *,buf,unsigned int,count)
```

In the latter case, the macro yields the following code:

```
int write(int fd,const char * buf,unsigned int count)
{
    long __res;
    asm("int $0x80"
        : "=a" (__res)
        : "0" (__NR_write), "b" ((long)fd),
          "c" ((long)buf), "d" ((long)count));
    if ((unsigned long)__res >= (unsigned long)-129) {
        errno = -__res;
        __res = -1;
    }
    return (int) __res;
}
```

The `__NR_write` macro is derived from the second parameter of `_syscall3`; it expands into the system call number of `write( )`. When compiling the preceding function, the following assembly language code is produced:

```
write:
     pushl %ebx              ; push ebx into stack
     movl 8(%esp), %ebx      ; put first parameter in ebx
     movl 12(%esp), %ecx     ; put second parameter in ecx
     movl 16(%esp), %edx     ; put third parameter in edx
     movl $4, %eax           ; put __NR_write in eax
     int $0x80               ; invoke system call
     cmpl $-125, %eax        ; check return code
     jbe .L1                 ; if no error, jump
     negl %eax               ; complement the value of eax
     movl %eax, errno        ; put result in errno
     movl $-1, %eax          ; set eax to -1
.L1: popl %ebx               ; pop ebx from stack
     ret                     ; return to calling program
```

Notice how the parameters of the `write( )` function are loaded into the CPU registers before the `int $0x80` instruction is executed. The value returned in `eax` must be interpreted as an error code if it lies between –1 and –129 (the kernel assumes that the largest error code defined in *include/generic/errno.h* is 129). If this is the case, the wrapper routine stores the value of `-eax` in `errno` and returns the value –1; otherwise, it returns the value of `eax`.

CHAPTER 12

The Virtual Filesystem

One of Linux's keys to success is its ability to coexist comfortably with other systems. You can transparently mount disks or partitions that host file formats used by Windows, other Unix systems, or even systems with tiny market shares like the Amiga. Linux manages to support multiple filesystem types in the same way other Unix variants do, through a concept called the Virtual Filesystem.

The idea behind the Virtual Filesystem is to put a wide range of information in the kernel to represent many different types of filesystems; there is a field or function to support each operation provided by all real filesystems supported by Linux. For each read, write, or other function called, the kernel substitutes the actual function that supports a native Linux filesystem, the NTFS filesystem, or whatever other filesystem the file is on.

This chapter discusses the aims, structure, and implementation of Linux's Virtual Filesystem. It focuses on three of the five standard Unix file types—namely, regular files, directories, and symbolic links. Device files are covered in Chapter 13, while pipes are discussed in Chapter 19. To show how a real filesystem works, Chapter 18 covers the Second Extended Filesystem that appears on nearly all Linux systems.

The Role of the Virtual Filesystem (VFS)

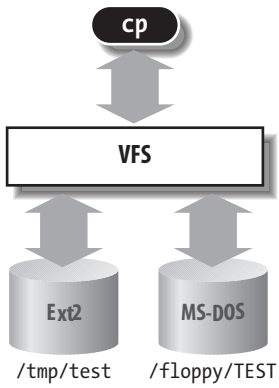
The *Virtual Filesystem* (also known as Virtual Filesystem Switch or *VFS*) is a kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.

For instance, let's assume that a user issues the shell command:

```
$ cp /floppy/TEST /tmp/test
```

where */floppy* is the mount point of an MS-DOS diskette and */tmp* is a normal Second Extended Filesystem (Ext2) directory. The VFS is an abstraction layer between the application program and the filesystem implementations (see Figure 12-1(a)). Therefore, the *cp* program is not required to know the filesystem types of */floppy/TEST* and */tmp/test*. Instead, *cp* interacts with the VFS by means of generic system

calls known to anyone who has done Unix programming (see the section "File-Handling System Calls" in Chapter 1); the code executed by *cp* is shown in Figure 12-1(b).




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

Figure 12-1. VFS role in a simple file copy operation

Filesystems supported by the VFS may be grouped into three main classes:

Disk-based filesystems
: These manage memory space available in a local disk or in some other device that emulates a disk (such as a USB flash drive). Some of the well-known disk-based filesystems supported by the VFS are:

  - Filesystems for Linux such as the widely used Second Extended Filesystem (Ext2), the recent Third Extended Filesystem (Ext3), and the Reiser Filesystems (ReiserFS)*
  - Filesystems for Unix variants such as sysv filesystem (System V, Coherent, Xenix), UFS (BSD, Solaris, NEXTSTEP), MINIX filesystem, and VERITAS VxFS (SCO UnixWare)
  - Microsoft filesystems such as MS-DOS, VFAT (Windows 95 and later releases), and NTFS (Windows NT 4 and later releases)
  - ISO9660 CD-ROM filesystem (formerly High Sierra Filesystem) and Universal Disk Format (UDF) DVD filesystem
  - Other proprietary filesystems such as IBM's OS/2 (HPFS), Apple's Macintosh (HFS), Amiga's Fast Filesystem (AFFS), and Acorn Disk Filing System (ADFS)

* Although these filesystems owe their birth to Linux, they have been ported to several other operating systems.

- Additional journaling filesystems originating in systems other than Linux such as IBM's JFS and SGI's XFS

Network filesystems
: These allow easy access to files included in filesystems belonging to other networked computers. Some well-known network filesystems supported by the VFS are NFS, Coda, AFS (Andrew filesystem), CIFS (Common Internet File System, used in Microsoft Windows), and NCP (Novell's NetWare Core Protocol).

Special filesystems
: These do not manage disk space, either locally or remotely. The */proc* filesystem is a typical example of a special filesystem (see the later section "Special Filesystems").

In this book, we describe in detail the Ext2 and Ext3 filesystems only (see Chapter 18); the other filesystems are not covered for lack of space.

As mentioned in the section "An Overview of the Unix Filesystem" in Chapter 1, Unix directories build a tree whose root is the / directory. The root directory is contained in the *root filesystem*, which in Linux, is usually of type Ext2 or Ext3. All other filesystems can be "mounted" on subdirectories of the root filesystem.*

A disk-based filesystem is usually stored in a hardware block device such as a hard disk, a floppy, or a CD-ROM. A useful feature of Linux's VFS allows it to handle *virtual block devices* such as */dev/loop0*, which may be used to mount filesystems stored in regular files. As a possible application, a user may protect her own private filesystem by storing an encrypted version of it in a regular file.

The first Virtual Filesystem was included in Sun Microsystems's SunOS in 1986. Since then, most Unix filesystems include a VFS. Linux's VFS, however, supports the widest range of filesystems.

The Common File Model

The key idea behind the VFS consists of introducing a *common file model* capable of representing all supported filesystems. This model strictly mirrors the file model provided by the traditional Unix filesystem. This is not surprising, because Linux wants to run its native filesystem with minimum overhead. However, each specific filesystem implementation must translate its physical organization into the VFS's common file model.

* When a filesystem is mounted on a directory, the contents of the directory in the parent filesystem are no longer accessible, because every pathname, including the mount point, will refer to the mounted filesystem. However, the original directory's content shows up again when the filesystem is unmounted. This somewhat surprising feature of Unix filesystems is used by system administrators to hide files; they simply mount a filesystem on the directory containing the files to be hidden.

For instance, in the common file model, each directory is regarded as a file, which contains a list of files and other directories. However, several non-Unix disk-based filesystems use a File Allocation Table (FAT), which stores the position of each file in the directory tree. In these filesystems, directories are not files. To stick to the VFS's common file model, the Linux implementations of such FAT-based filesystems must be able to construct on the fly, when needed, the files corresponding to the directories. Such files exist only as objects in kernel memory.

More essentially, the Linux kernel cannot hardcode a particular function to handle an operation such as `read()` or `ioctl()`. Instead, it must use a pointer for each operation; the pointer is made to point to the proper function for the particular filesystem being accessed.

Let's illustrate this concept by showing how the `read()` shown in Figure 12-1 would be translated by the kernel into a call specific to the MS-DOS filesystem. The application's call to `read()` makes the kernel invoke the corresponding `sys_read()` service routine, like every other system call. The file is represented by a `file` data structure in kernel memory, as we'll see later in this chapter. This data structure contains a field called `f_op` that contains pointers to functions specific to MS-DOS files, including a function that reads a file. `sys_read()` finds the pointer to this function and invokes it. Thus, the application's `read()` is turned into the rather indirect call:

```
file->f_op->read(...);
```

Similarly, the `write()` operation triggers the execution of a proper Ext2 write function associated with the output file. In short, the kernel is responsible for assigning the right set of pointers to the `file` variable associated with each open file, and then for invoking the call specific to each filesystem that the `f_op` field points to.

One can think of the common file model as object-oriented, where an *object* is a software construct that defines both a data structure and the methods that operate on it. For reasons of efficiency, Linux is not coded in an object-oriented language such as C++. Objects are therefore implemented as plain C data structures with some fields pointing to functions that correspond to the object's methods.

The common file model consists of the following object types:

The superblock object
: Stores information concerning a mounted filesystem. For disk-based filesystems, this object usually corresponds to a *filesystem control block* stored on disk.

The inode object
: Stores general information about a specific file. For disk-based filesystems, this object usually corresponds to a *file control block* stored on disk. Each inode object is associated with an *inode number*, which uniquely identifies the file within the filesystem.

The file object
: Stores information about the interaction between an open file and a process. This information exists only in kernel memory during the period when a process has the file open.

The dentry object
: Stores information about the linking of a directory entry (that is, a particular name of the file) with the corresponding file. Each disk-based filesystem stores this information in its own particular way on disk.

Figure 12-2 illustrates with a simple example how processes interact with files. Three different processes have opened the same file, two of them using the same hard link. In this case, each of the three processes uses its own file object, while only two dentry objects are required—one for each hard link. Both dentry objects refer to the same inode object, which identifies the superblock object and, together with the latter, the common disk file.

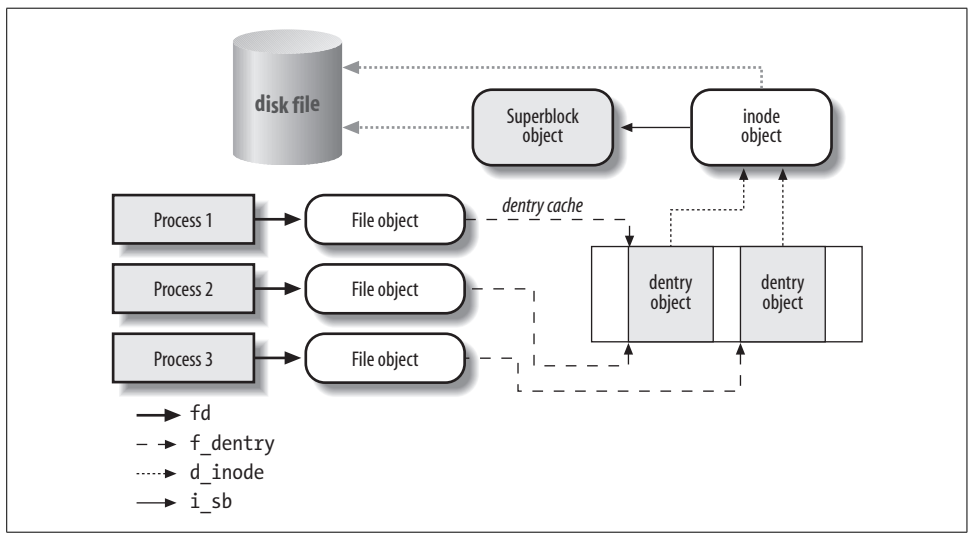


Figure 12-2. Interaction between processes and VFS objects

Besides providing a common interface to all filesystem implementations, the VFS has another important role related to system performance. The most recently used dentry objects are contained in a disk cache named the *dentry cache*, which speeds up the translation from a file pathname to the inode of the last pathname component.

Generally speaking, a *disk cache* is a software mechanism that allows the kernel to keep in RAM some information that is normally stored on a disk, so that further accesses to that data can be quickly satisfied without a slow access to the disk itself.

Notice how a disk cache differs from a hardware cache or a memory cache, neither of which has anything to do with disks or other devices. A hardware cache is a fast

static RAM that speeds up requests directed to the slower dynamic RAM (see the section "Hardware Cache" in Chapter 2). A memory cache is a software mechanism introduced to bypass the Kernel Memory Allocator (see the section "The Slab Allocator" in Chapter 8).

Beside the dentry cache and the inode cache, Linux uses other disk caches. The most important one, called the page cache, is described in detail in Chapter 15.

System Calls Handled by the VFS

Table 12-1 illustrates the VFS system calls that refer to filesystems, regular files, directories, and symbolic links. A few other system calls handled by the VFS, such as `ioperm()`, `ioctl()`, `pipe()`, and `mknod()`, refer to device files and pipes. These are discussed in later chapters. A last group of system calls handled by the VFS, such as `socket()`, `connect()`, and `bind()`, refer to sockets and are used to implement networking. Some of the kernel service routines that correspond to the system calls listed in Table 12-1 are discussed either in this chapter or in Chapter 18.

Table 12-1. Some system calls handled by the VFS

System call name	Description
`mount()` `umount()` `umount2()`	Mount/unmount filesystems
`sysfs()`	Get filesystem information
`statfs()` `fstatfs()` `statfs64()` `fstatfs64()` `ustat()`	Get filesystem statistics
`chroot()` `pivot_root()`	Change root directory
`chdir()` `fchdir()` `getcwd()`	Manipulate current directory
`mkdir()` `rmdir()`	Create and destroy directories
`getdents()` `getdents64()` `readdir()` `link()` `unlink()` `rename()` `lookup_dcookie()`	Manipulate directory entries
`readlink()` `symlink()`	Manipulate soft links
`chown()` `fchown()` `lchown()` `chown16()` `fchown16()` `lchown16()`	Modify file owner
`chmod()` `fchmod()` `utime()`	Modify file attributes
`stat()` `fstat()` `lstat()` `access()` `oldstat()` `oldfstat()` `oldlstat()` `stat64()` `lstat64()` `fstat64()`	Read file status
`open()` `close()` `creat()` `umask()`	Open, close, and create files
`dup()` `dup2()` `fcntl()` `fcntl64()`	Manipulate file descriptors
`select()` `poll()`	Wait for events on a set of file descriptors
`truncate()` `ftruncate()` `truncate64()` `ftruncate64()`	Change file size
`lseek()` `_llseek()`	Change file pointer
`read()` `write()` `readv()` `writev()` `sendfile()` `sendfile64()` `readahead()`	Carry out file I/O operations

Table 12-1. Some system calls handled by the VFS (continued)

System call name	Description
`io_setup()` `io_submit()` `io_getevents()` `io_cancel()` `io_destroy()`	Asynchronous I/O (allows multiple outstanding read and write requests)
`pread64()` `pwrite64()`	Seek file and access it
`mmap()` `mmap2()` `munmap()` `madvise()` `mincore()` `remap_file_pages()`	Handle file memory mapping
`fdatasync()` `fsync()` `sync()` `msync()`	Synchronize file data
`flock()`	Manipulate file lock
`setxattr()` `lsetxattr()` `fsetxattr()` `getxattr()` `lgetxattr()` `fgetxattr()` `listxattr()` `llistxattr()` `flistxattr()` `removexattr()` `lremovexattr()` `fremovexattr()`	Manipulate file extended attributes

We said earlier that the VFS is a layer between application programs and specific 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 it does not have to invoke a specific filesystem procedure. In some sense, the VFS could be considered a "generic" filesystem that relies, when necessary, on specific ones.

VFS Data Structures

Each VFS object is stored in a suitable data structure, which includes both the object attributes and a pointer to a table of object methods. The kernel may dynamically modify the methods of the object and, hence, it may install specialized behavior for the object. The following sections explain the VFS objects and their interrelationships in detail.

Superblock Objects

A superblock object consists of a `super_block` structure whose fields are described in Table 12-2.

Table 12-2. The fields of the superblock object

Type	Field	Description
`struct list_head`	`s_list`	Pointers for superblock list
`dev_t`	`s_dev`	Device identifier

Table 12-2. The fields of the superblock object (continued)

Type	Field	Description
unsigned long	s_blocksize	Block size in bytes
unsigned long	s_old_blocksize	Block size in bytes as reported by the underlying block device driver
unsigned char	s_blocksize_bits	Block size in number of bits
unsigned char	s_dirt	Modified (dirty) flag
unsigned long long	s_maxbytes	Maximum size of the files
struct file_system_type *	s_type	Filesystem type
struct super_operations *	s_op	Superblock methods
struct dquot_operations *	dq_op	Disk quota handling methods
struct quotactl_ops *	s_qcop	Disk quota administration methods
struct export_operations *	s_export_op	Export operations used by network filesystems
unsigned long	s_flags	Mount flags
unsigned long	s_magic	Filesystem magic number
struct dentry *	s_root	Dentry object of the filesystem's root directory
struct rw_semaphore	s_umount	Semaphore used for unmounting
struct semaphore	s_lock	Superblock semaphore
int	s_count	Reference counter
int	s_syncing	Flag indicating that inodes of the superblock are being synchronized
int	s_need_sync_fs	Flag used when synchronizing the superblock's mounted filesystem
atomic_t	s_active	Secondary reference counter
void *	s_security	Pointer to superblock security structure
struct xattr_handler **	s_xattr	Pointer to superblock extended attribute structure
struct list_head	s_inodes	List of all inodes
struct list_head	s_dirty	List of modified inodes
struct list_head	s_io	List of inodes waiting to be written to disk
struct hlist_head	s_anon	List of anonymous dentries for handling remote network filesystems
struct list_head	s_files	List of file objects
struct block_device *	s_bdev	Pointer to the block device driver descriptor
struct list_head	s_instances	Pointers for a list of superblock objects of a given filesystem type (see the later section "Filesystem Type Registration")

Table 12-2. The fields of the superblock object (continued)

Type	Field	Description
`struct quota_info`	`s_dquot`	Descriptor for disk quota
`int`	`s_frozen`	Flag used when freezing the filesystem (forcing it to a consistent state)
`wait_queue_head_t`	`s_wait_unfrozen`	Wait queue where processes sleep until the filesystem is unfrozen
`char[]`	`s_id`	Name of the block device containing the superblock
`void *`	`s_fs_info`	Pointer to superblock information of a specific filesystem
`struct semaphore`	`s_vfs_rename_sem`	Semaphore used by VFS when renaming files across directories
`u32`	`s_time_gran`	Timestamp's granularity (in nanoseconds)

All superblock objects are linked in a circular doubly linked list. The first element of this list is represented by the `super_blocks` variable, while the `s_list` field of the superblock object stores the pointers to the adjacent elements in the list. The `sb_lock` spin lock protects the list against concurrent accesses in multiprocessor systems.

The `s_fs_info` field points to superblock information that belongs to a specific filesystem; for instance, as we'll see later in Chapter 18, if the superblock object refers to an Ext2 filesystem, the field points to an `ext2_sb_info` structure, which includes the disk allocation bit masks and other data of no concern to the VFS common file model.

In general, data pointed to by the `s_fs_info` field is information from the disk duplicated in memory for reasons of efficiency. Each disk-based filesystem needs to access and update its allocation bitmaps in order to allocate or release disk blocks. The VFS allows these filesystems to act directly on the `s_fs_info` field of the superblock in memory without accessing the disk.

This approach leads to a new problem, however: the VFS superblock might end up no longer synchronized with the corresponding superblock on disk. It is thus necessary to introduce an `s_dirt` flag, which specifies whether the superblock is dirty—that is, whether the data on the disk must be updated. The lack of synchronization leads to the familiar problem of a corrupted filesystem when a site's power goes down without giving the user the chance to shut down a system cleanly. As we'll see in the section "Writing Dirty Pages to Disk" in Chapter 15, Linux minimizes this problem by periodically copying all dirty superblocks to disk.

The methods associated with a superblock are called *superblock operations*. They are described by the `super_operations` structure whose address is included in the `s_op` field.

Each specific filesystem can define its own superblock operations. When the VFS needs to invoke one of them, say `read_inode( )`, it executes the following:

```
sb->s_op->read_inode(inode);
```

where sb stores the address of the superblock object involved. The read_inode field of the super_operations table contains the address of the suitable function, which is therefore directly invoked.

Let's briefly describe the superblock operations, which implement higher-level operations like deleting files or mounting disks. They are listed in the order they appear in the super_operations table:

alloc_inode(sb)
: Allocates space for an inode object, including the space required for filesystem-specific data.

destroy_inode(inode)
: Destroys an inode object, including the filesystem-specific data.

read_inode(inode)
: Fills the fields of the inode object passed as the parameter with the data on disk; the i_ino field of the inode object identifies the specific filesystem inode on the disk to be read.

dirty_inode(inode)
: Invoked when the inode is marked as modified (dirty). Used by filesystems such as ReiserFS and Ext3 to update the filesystem journal on disk.

write_inode(inode, flag)
: Updates a filesystem inode with the contents of the inode object passed as the parameter; the i_ino field of the inode object identifies the filesystem inode on disk that is concerned. The flag parameter indicates whether the I/O operation should be synchronous.

put_inode(inode)
: Invoked when the inode is released—its reference counter is decreased—to perform filesystem-specific operations.

drop_inode(inode)
: Invoked when the inode is about to be destroyed—that is, when the last user releases the inode; filesystems that implement this method usually make use of generic_drop_inode(). This function removes every reference to the inode from the VFS data structures and, if the inode no longer appears in any directory, invokes the delete_inode superblock method to delete the inode from the filesystem.

delete_inode(inode)
: Invoked when the inode must be destroyed. Deletes the VFS inode in memory and the file data and metadata on disk.

put_super(super)
: Releases the superblock object passed as the parameter (because the corresponding filesystem is unmounted).

write_super(super)
: Updates a filesystem superblock with the contents of the object indicated.

`sync_fs(sb, wait)`
: Invoked when flushing the filesystem to update filesystem-specific data structures on disk (used by journaling filesystems).

`write_super_lockfs(super)`
: Blocks changes to the filesystem and updates the superblock with the contents of the object indicated. This method is invoked when the filesystem is frozen, for instance by the Logical Volume Manager (LVM) driver.

`unlockfs(super)`
: Undoes the block of filesystem updates achieved by the `write_super_lockfs` superblock method.

`statfs(super, buf)`
: Returns statistics on a filesystem by filling the `buf` buffer.

`remount_fs(super, flags, data)`
: Remounts the filesystem with new options (invoked when a mount option must be changed).

`clear_inode(inode)`
: Invoked when a disk inode is being destroyed to perform filesystem-specific operations.

`umount_begin(super)`
: Aborts a mount operation because the corresponding unmount operation has been started (used only by network filesystems).

`show_options(seq_file, vfsmount)`
: Used to display the filesystem-specific options

`quota_read(super, type, data, size, offset)`
: Used by the quota system to read data from the file that specifies the limits for this filesystem.*

`quota_write(super, type, data, size, offset)`
: Used by the quota system to write data into the file that specifies the limits for this filesystem.

The preceding methods are available to all possible filesystem types. However, only a subset of them applies to each specific filesystem; the fields corresponding to unimplemented methods are set to `NULL`. Notice that no `get_super` method to read a superblock is defined—how could the kernel invoke a method of an object yet to be read from disk? We'll find an equivalent `get_sb` method in another object describing the filesystem type (see the later section "Filesystem Type Registration").

* The *quota system* defines for each user and/or group limits on the amount of space that can be used on a given filesystem (see the `quotactl()` system call.)

Inode Objects

All information needed by the filesystem to handle a file is included in a data structure called an inode. A filename is a casually assigned label that can be changed, but the inode is unique to the file and remains the same as long as the file exists. An inode object in memory consists of an `inode` structure whose fields are described in Table 12-3.

Table 12-3. The fields of the inode object

Type	Field	Description
`struct hlist_node`	`i_hash`	Pointers for the hash list
`struct list_head`	`i_list`	Pointers for the list that describes the inode's current state
`struct list_head`	`i_sb_list`	Pointers for the list of inodes of the superblock
`struct list_head`	`i_dentry`	The head of the list of dentry objects referencing this inode
`unsigned long`	`i_ino`	inode number
`atomic_t`	`i_count`	Usage counter
`umode_t`	`i_mode`	File type and access rights
`unsigned int`	`i_nlink`	Number of hard links
`uid_t`	`i_uid`	Owner identifier
`gid_t`	`i_gid`	Group identifier
`dev_t`	`i_rdev`	Real device identifier
`loff_t`	`i_size`	File length in bytes
`struct timespec`	`i_atime`	Time of last file access
`struct timespec`	`i_mtime`	Time of last file write
`struct timespec`	`i_ctime`	Time of last inode change
`unsigned int`	`i_blkbits`	Block size in number of bits
`unsigned long`	`i_blksize`	Block size in bytes
`unsigned long`	`i_version`	Version number, automatically increased after each use
`unsigned long`	`i_blocks`	Number of blocks of the file
`unsigned short`	`i_bytes`	Number of bytes in the last block of the file
`unsigned char`	`i_sock`	Nonzero if file is a socket
`spinlock_t`	`i_lock`	Spin lock protecting some fields of the inode
`struct semaphore`	`i_sem`	inode semaphore
`struct rw_semaphore`	`i_alloc_sem`	Read/write semaphore protecting against race conditions in direct I/O file operations
`struct inode_operations *`	`i_op`	inode operations
`struct file_operations *`	`i_fop`	Default file operations
`struct super_block *`	`i_sb`	Pointer to superblock object

Table 12-3. The fields of the inode object (continued)

Type	Field	Description
`struct file_lock *`	`i_flock`	Pointer to file lock list
`struct address_space *`	`i_mapping`	Pointer to an `address_space` object (see Chapter 15)
`struct address_space`	`i_data`	`address_space` object of the file
`struct dquot * []`	`i_dquot`	inode disk quotas
`struct list_head`	`i_devices`	Pointers for a list of inodes relative to a specific character or block device (see Chapter 13)
`struct pipe_inode_info *`	`i_pipe`	Used if the file is a pipe (see Chapter 19)
`struct block_device *`	`i_bdev`	Pointer to the block device driver
`struct cdev *`	`i_cdev`	Pointer to the character device driver
`int`	`i_cindex`	Index of the device file within a group of minor numbers
`__u32`	`i_generation`	inode version number (used by some filesystems)
`unsigned long`	`i_dnotify_mask`	Bit mask of directory notify events
`struct dnotify_struct *`	`i_dnotify`	Used for directory notifications
`unsigned long`	`i_state`	inode state flags
`unsigned long`	`dirtied_when`	Dirtying time (in ticks) of the inode
`unsigned int`	`i_flags`	Filesystem mount flags
`atomic_t`	`i_writecount`	Usage counter for writing processes
`void *`	`i_security`	Pointer to inode's security structure
`void *`	`u.generic_ip`	Pointer to private data
`seqcount_t`	`i_size_seqcount`	Sequence counter used in SMP systems to get consistent values for `i_size`

Each inode object duplicates some of the data included in the disk inode—for instance, the number of blocks allocated to the file. When the value of the `i_state` field is equal to `I_DIRTY_SYNC`, `I_DIRTY_DATASYNC`, or `I_DIRTY_PAGES`, the inode is dirty—that is, the corresponding disk inode must be updated. The `I_DIRTY` macro can be used to check the value of these three flags at once (see later for details). Other values of the `i_state` field are `I_LOCK` (the inode object is involved in an I/O transfer), `I_FREEING` (the inode object is being freed), `I_CLEAR` (the inode object contents are no longer meaningful), and `I_NEW` (the inode object has been allocated but not yet filled with data read from the disk inode).

Each inode object always appears in one of the following circular doubly linked lists (in all cases, the pointers to the adjacent elements are stored in the `i_list` field):

- The list of valid unused inodes, typically those mirroring valid disk inodes and not currently used by any process. These inodes are not dirty and their `i_count` field is set to 0. The first and last elements of this list are referenced by the `next` and `prev` fields, respectively, of the `inode_unused` variable. This list acts as a disk cache.

- The list of in-use inodes, that is, those mirroring valid disk inodes and used by some process. These inodes are not dirty and their `i_count` field is positive. The first and last elements are referenced by the `inode_in_use` variable.
- The list of dirty inodes. The first and last elements are referenced by the `s_dirty` field of the corresponding superblock object.

Each of the lists just mentioned links the `i_list` fields of the proper inode objects.

Moreover, each inode object is also included in a per-filesystem doubly linked circular list headed at the `s_inodes` field of the superblock object; the `i_sb_list` field of the inode object stores the pointers for the adjacent elements in this list.

Finally, the inode objects are also included in a hash table named `inode_hashtable`. The hash table speeds up the search of the inode object when the kernel knows both the inode number and the address of the superblock object corresponding to the filesystem that includes the file. Because hashing may induce collisions, the inode object includes an `i_hash` field that contains a backward and a forward pointer to other inodes that hash to the same position; this field creates a doubly linked list of those inodes.

The methods associated with an inode object are also called *inode operations*. They are described by an `inode_operations` structure, whose address is included in the `i_op` field. Here are the inode operations in the order they appear in the `inode_operations` table:

`create(dir, dentry, mode, nameidata)`
: Creates a new disk inode for a regular file associated with a dentry object in some directory.

`lookup(dir, dentry, nameidata)`
: Searches a directory for an inode corresponding to the filename included in a dentry object.

`link(old_dentry, dir, new_dentry)`
: Creates a new hard link that refers to the file specified by `old_dentry` in the directory `dir`; the new hard link has the name specified by `new_dentry`.

`unlink(dir, dentry)`
: Removes the hard link of the file specified by a dentry object from a directory.

`symlink(dir, dentry, symname)`
: Creates a new inode for a symbolic link associated with a dentry object in some directory.

`mkdir(dir, dentry, mode)`
: Creates a new inode for a directory associated with a dentry object in some directory.

`rmdir(dir, dentry)`
: Removes from a directory the subdirectory whose name is included in a dentry object.

`mknod(dir, dentry, mode, rdev)`
: Creates a new disk inode for a special file associated with a dentry object in some directory. The `mode` and `rdev` parameters specify, respectively, the file type and the device's major and minor numbers.

`rename(old_dir, old_dentry, new_dir, new_dentry)`
: Moves the file identified by `old_entry` from the `old_dir` directory to the `new_dir` one. The new filename is included in the dentry object that `new_dentry` points to.

`readlink(dentry, buffer, buflen)`
: Copies into a User Mode memory area specified by `buffer` the file pathname corresponding to the symbolic link specified by the dentry.

`follow_link(inode, nameidata)`
: Translates a symbolic link specified by an inode object; if the symbolic link is a relative pathname, the lookup operation starts from the directory specified in the second parameter.

`put_link(dentry, nameidata)`
: Releases all temporary data structures allocated by the `follow_link` method to translate a symbolic link.

`truncate(inode)`
: Modifies the size of the file associated with an inode. Before invoking this method, it is necessary to set the `i_size` field of the inode object to the required new size.

`permission(inode, mask, nameidata)`
: Checks whether the specified access mode is allowed for the file associated with `inode`.

`setattr(dentry, iattr)`
: Notifies a "change event" after touching the inode attributes.

`getattr(mnt, dentry, kstat)`
: Used by some filesystems to read inode attributes.

`setxattr(dentry, name, value, size, flags)`
: Sets an "extended attribute" of an inode (extended attributes are stored on disk blocks outside of any inode).

`getxattr(dentry, name, buffer, size)`
: Gets an extended attribute of an inode.

`listxattr(dentry, buffer, size)`
: Gets the whole list of extended attribute names.

`removexattr(dentry, name)`
: Removes an extended attribute of an inode.

The methods just listed are available to all possible inodes and filesystem types. However, only a subset of them applies to a specific inode and filesystem; the fields corresponding to unimplemented methods are set to `NULL`.

File Objects

A file object describes how a process interacts with a file it has opened. The object is created when the file is opened and consists of a `file` structure, whose fields are described in Table 12-4. Notice that file objects have no corresponding image on disk, and hence no "dirty" field is included in the `file` structure to specify that the file object has been modified.

Table 12-4. The fields of the file object

Type	Field	Description
`struct list_head`	`f_list`	Pointers for generic file object list
`struct dentry *`	`f_dentry`	dentry object associated with the file
`struct vfsmount *`	`f_vfsmnt`	Mounted filesystem containing the file
`struct file_operations *`	`f_op`	Pointer to file operation table
`atomic_t`	`f_count`	File object's reference counter
`unsigned int`	`f_flags`	Flags specified when opening the file
`mode_t`	`f_mode`	Process access mode
`int`	`f_error`	Error code for network write operation
`loff_t`	`f_pos`	Current file offset (file pointer)
`struct fown_struct`	`f_owner`	Data for I/O event notification via signals
`unsigned int`	`f_uid`	User's UID
`unsigned int`	`f_gid`	User group ID
`struct file_ra_state`	`f_ra`	File read-ahead state (see Chapter 16)
`size_t`	`f_maxcount`	Maximum number of bytes that can be read or written with a single operation (currently set to $2^{31}-1$)
`unsigned long`	`f_version`	Version number, automatically increased after each use
`void *`	`f_security`	Pointer to file object's security structure
`void *`	`private_data`	Pointer to data specific for a filesystem or a device driver
`struct list_head`	`f_ep_links`	Head of the list of event poll waiters for this file
`spinlock_t`	`f_ep_lock`	Spin lock protecting the `f_ep_links` list
`struct address_space *`	`f_mapping`	Pointer to file's address space object (see Chapter 15)

The main information stored in a file object is the *file pointer*—the current position in the file from which the next operation will take place. Because several processes may access the same file concurrently, the file pointer must be kept in the file object rather than the inode object.

File objects are allocated through a slab cache named *filp*, whose descriptor address is stored in the `filp_cachep` variable. Because there is a limit on the number of file objects that can be allocated, the `files_stat` variable specifies in the `max_files` field

the maximum number of allocatable file objects—i.e., the maximum number of files that can be accessed at the same time in the system.*

"In use" file objects are collected in several lists rooted at the superblocks of the owning filesystems. Each superblock object stores in the `s_files` field the head of a list of file objects; thus, file objects of files belonging to different filesystems are included in different lists. The pointers to the previous and next element in the list are stored in the `f_list` field of the file object. The `files_lock` spin lock protects the superblock `s_files` lists against concurrent accesses in multiprocessor systems.

The `f_count` field of the file object is a reference counter: it counts the number of processes that are using the file object (remember that lightweight processes created with the `CLONE_FILES` flag share the table that identifies the open files, thus they use the same file objects). The counter is also increased when the file object is used by the kernel itself—for instance, when the object is inserted in a list, or when a `dup()` system call has been issued.

When the VFS must open a file on behalf of a process, it invokes the `get_empty_filp( )` function to allocate a new file object. The function invokes `kmem_cache_alloc()` to get a free file object from the *filp* cache, then it initializes the fields of the object as follows:

```
memset(f, 0, sizeof(*f));
INIT_LIST_HEAD(&f->f_ep_links);
spin_lock_init(&f->f_ep_lock);
atomic_set(&f->f_count, 1);
f->f_uid = current->fsuid;
f->f_gid = current->fsgid;
f->f_owner.lock = RW_LOCK_UNLOCKED;
INIT_LIST_HEAD(&f->f_list);
f->f_maxcount = INT_MAX;
```

As we explained earlier in the section "The Common File Model," each filesystem includes its own set of *file operations* that perform such activities as reading and writing a file. When the kernel loads an inode into memory from disk, it stores a pointer to these file operations in a `file_operations` structure whose address is contained in the `i_fop` field of the inode object. When a process opens the file, the VFS initializes the `f_op` field of the new file object with the address stored in the inode so that further calls to file operations can use these functions. If necessary, the VFS may later modify the set of file operations by storing a new value in `f_op`.

The following list describes the file operations in the order in which they appear in the `file_operations` table:

* The `files_init()` function, executed during kernel initialization, sets the `max_files` field to one-tenth of the available RAM in kilobytes, but the system administrator can tune this parameter by writing into the */proc/sys/fs/file-max* file. Moreover, the superuser can always get a file object, even if `max_files` file objects have already been allocated.

`llseek(file, offset, origin)`
: Updates the file pointer.

`read(file, buf, count, offset)`
: Reads `count` bytes from a file starting at position `*offset`; the value `*offset` (which usually corresponds to the file pointer) is then increased.

`aio_read(req, buf, len, pos)`
: Starts an asynchronous I/O operation to read `len` bytes into `buf` from file position `pos` (introduced to support the `io_submit()` system call).

`write(file, buf, count, offset)`
: Writes `count` bytes into a file starting at position `*offset`; the value `*offset` (which usually corresponds to the file pointer) is then increased.

`aio_write(req, buf, len, pos)`
: Starts an asynchronous I/O operation to write `len` bytes from `buf` to file position `pos`.

`readdir(dir, dirent, filldir)`
: Returns the next directory entry of a directory in `dirent`; the `filldir` parameter contains the address of an auxiliary function that extracts the fields in a directory entry.

`poll(file, poll_table)`
: Checks whether there is activity on a file and goes to sleep until something happens on it.

`ioctl(inode, file, cmd, arg)`
: Sends a command to an underlying hardware device. This method applies only to device files.

`unlocked_ioctl(file, cmd, arg)`
: Similar to the `ioctl` method, but it does not take the big kernel lock (see the section "The Big Kernel Lock" in Chapter 5). It is expected that all device drivers and all filesystems will implement this new method instead of the `ioctl` method.

`compat_ioctl(file, cmd, arg)`
: Method used to implement the `ioctl()` 32-bit system call by 64-bit kernels.

`mmap(file, vma)`
: Performs a memory mapping of the file into a process address space (see the section "Memory Mapping" in Chapter 16).

`open(inode, file)`
: Opens a file by creating a new file object and linking it to the corresponding inode object (see the section "The open( ) System Call" later in this chapter).

`flush(file)`
: Called when a reference to an open file is closed. The actual purpose of this method is filesystem-dependent.

`release(inode, file)`
: Releases the file object. Called when the last reference to an open file is closed—that is, when the `f_count` field of the file object becomes 0.

`fsync(file, dentry, flag)`
: Flushes the file by writing all cached data to disk.

`aio_fsync(req, flag)`
: Starts an asynchronous I/O flush operation.

`fasync(fd, file, on)`
: Enables or disables I/O event notification by means of signals.

`lock(file, cmd, file_lock)`
: Applies a lock to the file (see the section "File Locking" later in this chapter).

`readv(file, vector, count, offset)`
: Reads bytes from a file and puts the results in the buffers described by `vector`; the number of buffers is specified by `count`.

`writev(file, vector, count, offset)`
: Writes bytes into a file from the buffers described by `vector`; the number of buffers is specified by `count`.

`sendfile(in_file, offset, count, file_send_actor, out_file)`
: Transfers data from `in_file` to `out_file` (introduced to support the `sendfile()` system call).

`sendpage(file, page, offset, size, pointer, fill)`
: Transfers data from `file` to the page cache's `page`; this is a low-level method used by `sendfile()` and by the networking code for sockets.

`get_unmapped_area(file, addr, len, offset, flags)`
: Gets an unused address range to map the file.

`check_flags(flags)`
: Method invoked by the service routine of the `fcntl()` system call to perform additional checks when setting the status flags of a file (`F_SETFL` command). Currently used only by the NFS network filesystem.

`dir_notify(file, arg)`
: Method invoked by the service routine of the `fcntl()` system call when establishing a directory change notification (`F_NOTIFY` command). Currently used only by the Common Internet File System (CIFS) network filesystem.

`flock(file, flag, lock)`
: Used to customize the behavior of the `flock()` system call. No official Linux filesystem makes use of this method.

The methods just described are available to all possible file types. However, only a subset of them apply to a specific file type; the fields corresponding to unimplemented methods are set to `NULL`.

dentry Objects

We mentioned in the section "The Common File Model" that the VFS considers each directory a file that contains a list of files and other directories. We will discuss in Chapter 18 how directories are implemented on a specific filesystem. Once a directory entry is read into memory, however, it is transformed by the VFS into a dentry object based on the `dentry` structure, whose fields are described in Table 12-5. The kernel creates a dentry object for every component of a pathname that a process looks up; the dentry object associates the component to its corresponding inode. For example, when looking up the */tmp/test* pathname, the kernel creates a dentry object for the / root directory, a second dentry object for the *tmp* entry of the root directory, and a third dentry object for the *test* entry of the */tmp* directory.

Notice that dentry objects have no corresponding image on disk, and hence no field is included in the `dentry` structure to specify that the object has been modified. Dentry objects are stored in a slab allocator cache whose descriptor is `dentry_cache`; dentry objects are thus created and destroyed by invoking `kmem_cache_alloc()` and `kmem_cache_free()`.

Table 12-5. The fields of the dentry object

Type	Field	Description
`atomic_t`	`d_count`	Dentry object usage counter
`unsigned int`	`d_flags`	Dentry cache flags
`spinlock_t`	`d_lock`	Spin lock protecting the dentry object
`struct inode *`	`d_inode`	Inode associated with filename
`struct dentry *`	`d_parent`	Dentry object of parent directory
`struct qstr`	`d_name`	Filename
`struct list_head`	`d_lru`	Pointers for the list of unused dentries
`struct list_head`	`d_child`	For directories, pointers for the list of directory dentries in the same parent directory
`struct list_head`	`d_subdirs`	For directories, head of the list of subdirectory dentries
`struct list_head`	`d_alias`	Pointers for the list of dentries associated with the same inode (alias)
`unsigned long`	`d_time`	Used by `d_revalidate` method
`struct dentry_operations*`	`d_op`	Dentry methods
`struct super_block *`	`d_sb`	Superblock object of the file
`void *`	`d_fsdata`	Filesystem-dependent data
`struct rcu_head`	`d_rcu`	The RCU descriptor used when reclaiming the dentry object (see the section "Read-Copy Update (RCU)" in Chapter 5)
`struct dcookie_struct *`	`d_cookie`	Pointer to structure used by kernel profilers
`struct hlist_node`	`d_hash`	Pointer for list in hash table entry

Table 12-5. The fields of the dentry object (continued)

Type	Field	Description
int	d_mounted	For directories, counter for the number of filesystems mounted on this dentry
unsigned char[]	d_iname	Space for short filename

Each dentry object may be in one of four states:

Free
: The dentry object contains no valid information and is not used by the VFS. The corresponding memory area is handled by the slab allocator.

Unused
: The dentry object is not currently used by the kernel. The d_count usage counter of the object is 0, but the d_inode field still points to the associated inode. The dentry object contains valid information, but its contents may be discarded if necessary in order to reclaim memory.

In use
: The dentry object is currently used by the kernel. The d_count usage counter is positive, and the d_inode field points to the associated inode object. The dentry object contains valid information and cannot be discarded.

Negative
: The inode associated with the dentry does not exist, either because the corresponding disk inode has been deleted or because the dentry object was created by resolving a pathname of a nonexistent file. The d_inode field of the dentry object is set to NULL, but the object still remains in the dentry cache, so that further lookup operations to the same file pathname can be quickly resolved. The term "negative" is somewhat misleading, because no negative value is involved.

The methods associated with a dentry object are called *dentry operations*; they are described by the dentry_operations structure, whose address is stored in the d_op field. Although some filesystems define their own dentry methods, the fields are usually NULL and the VFS replaces them with default functions. Here are the methods, in the order they appear in the dentry_operations table:

d_revalidate(dentry, nameidata)
: Determines whether the dentry object is still valid before using it for translating a file pathname. The default VFS function does nothing, although network filesystems may specify their own functions.

d_hash(dentry, name)
: Creates a hash value; this function is a filesystem-specific hash function for the dentry hash table. The dentry parameter identifies the directory containing the component. The name parameter points to a structure containing both the pathname component to be looked up and the value produced by the hash function.

`d_compare(dir, name1, name2)`
: Compares two filenames; `name1` should belong to the directory referenced by `dir`. The default VFS function is a normal string match. However, each filesystem can implement this method in its own way. For instance, MS-DOS does not distinguish capital from lowercase letters.

`d_delete(dentry)`
: Called when the last reference to a dentry object is deleted (`d_count` becomes 0). The default VFS function does nothing.

`d_release(dentry)`
: Called when a dentry object is going to be freed (released to the slab allocator). The default VFS function does nothing.

`d_iput(dentry, ino)`
: Called when a dentry object becomes "negative"—that is, it loses its inode. The default VFS function invokes `iput( )` to release the inode object.

The dentry Cache

Because reading a directory entry from disk and constructing the corresponding dentry object requires considerable time, it makes sense to keep in memory dentry objects that you've finished with but might need later. For instance, people often edit a file and then compile it, or edit and print it, or copy it and then edit the copy. In such cases, the same file needs to be repeatedly accessed.

To maximize efficiency in handling dentries, Linux uses a dentry cache, which consists of two kinds of data structures:

- A set of dentry objects in the in-use, unused, or negative state.
- A hash table to derive the dentry object associated with a given filename and a given directory quickly. As usual, if the required object is not included in the dentry cache, the search function returns a null value.

The dentry cache also acts as a controller for an *inode cache*. The inodes in kernel memory that are associated with unused dentries are not discarded, because the dentry cache is still using them. Thus, the inode objects are kept in RAM and can be quickly referenced by means of the corresponding dentries.

All the "unused" dentries are included in a doubly linked "Least Recently Used" list sorted by time of insertion. In other words, the dentry object that was last released is put in front of the list, so the least recently used dentry objects are always near the end of the list. When the dentry cache has to shrink, the kernel removes elements from the tail of this list so that the most recently used objects are preserved. The addresses of the first and last elements of the LRU list are stored in the `next` and `prev` fields of the `dentry_unused` variable of type `list_head`. The `d_lru` field of the dentry object contains pointers to the adjacent dentries in the list.

Each "in use" dentry object is inserted into a doubly linked list specified by the `i_dentry` field of the corresponding inode object (because each inode could be associated with several hard links, a list is required). The `d_alias` field of the dentry object stores the addresses of the adjacent elements in the list. Both fields are of type `struct list_head`.

An "in use" dentry object may become "negative" when the last hard link to the corresponding file is deleted. In this case, the dentry object is moved into the LRU list of unused dentries. Each time the kernel shrinks the dentry cache, negative dentries move toward the tail of the LRU list so that they are gradually freed (see the section "Reclaiming Pages of Shrinkable Disk Caches" in Chapter 17).

The hash table is implemented by means of a `dentry_hashtable` array. Each element is a pointer to a list of dentries that hash to the same hash table value. The array's size usually depends on the amount of RAM installed in the system; the default value is 256 entries per megabyte of RAM. The `d_hash` field of the dentry object contains pointers to the adjacent elements in the list associated with a single hash value. The hash function produces its value from both the dentry object of the directory and the filename.

The `dcache_lock` spin lock protects the dentry cache data structures against concurrent accesses in multiprocessor systems. The `d_lookup()` function looks in the hash table for a given parent dentry object and filename; to avoid race conditions, it makes use of a seqlock (see the section "Seqlocks" in Chapter 5). The `__d_lookup()` function is similar, but it assumes that no race condition can happen, so it does not use the seqlock.

Files Associated with a Process

We mentioned in the section "An Overview of the Unix Filesystem" in Chapter 1 that each process has its own current working directory and its own root directory. These are only two examples of data that must be maintained by the kernel to represent the interactions between a process and a filesystem. A whole data structure of type `fs_struct` is used for that purpose (see Table 12-6), and each process descriptor has an `fs` field that points to the process `fs_struct` structure.

Table 12-6. The fields of the fs_struct structure

Type	Field	Description
`atomic_t`	`count`	Number of processes sharing this table
`rwlock_t`	`lock`	Read/write spin lock for the table fields
`int`	`umask`	Bit mask used when opening the file to set the file permissions
`struct dentry *`	`root`	Dentry of the root directory
`struct dentry *`	`pwd`	Dentry of the current working directory
`struct dentry *`	`altroot`	Dentry of the emulated root directory (always `NULL` for the 80 × 86 architecture)

Table 12-6. The fields of the fs_struct structure (continued)

Type	Field	Description
struct vfsmount *	rootmnt	Mounted filesystem object of the root directory
struct vfsmount *	pwdmnt	Mounted filesystem object of the current working directory
struct vfsmount *	altrootmnt	Mounted filesystem object of the emulated root directory (always NULL for the 80×86 architecture)

A second table, whose address is contained in the files field of the process descriptor, specifies which files are currently opened by the process. It is a files_struct structure whose fields are illustrated in Table 12-7.

Table 12-7. The fields of the files_struct structure

Type	Field	Description
atomic_t	count	Number of processes sharing this table
rwlock_t	file_lock	Read/write spin lock for the table fields
int	max_fds	Current maximum number of file objects
int	max_fdset	Current maximum number of file descriptors
int	next_fd	Maximum file descriptors ever allocated plus 1
struct file **	fd	Pointer to array of file object pointers
fd_set *	close_on_exec	Pointer to file descriptors to be closed on exec( )
fd_set *	open_fds	Pointer to open file descriptors
fd_set	close_on_exec_init	Initial set of file descriptors to be closed on exec( )
fd_set	open_fds_init	Initial set of file descriptors
struct file *[]	fd_array	Initial array of file object pointers

The fd field points to an array of pointers to file objects. The size of the array is stored in the max_fds field. Usually, fd points to the fd_array field of the files_struct structure, which includes 32 file object pointers. If the process opens more than 32 files, the kernel allocates a new, larger array of file pointers and stores its address in the fd fields; it also updates the max_fds field.

For every file with an entry in the fd array, the array index is the *file descriptor*. Usually, the first element (index 0) of the array is associated with the standard input of the process, the second with the standard output, and the third with the standard error (see Figure 12-3). Unix processes use the file descriptor as the main file identifier. Notice that, thanks to the dup( ), dup2( ), and fcntl( ) system calls, two file descriptors may refer to the same opened file—that is, two elements of the array could point to the same file object. Users see this all the time when they use shell constructs such as 2>&1 to redirect the standard error to the standard output.

A process cannot use more than NR_OPEN (usually, 1,048,576) file descriptors. The kernel also enforces a dynamic bound on the maximum number of file descriptors in

the signal->rlim[RLIMIT_NOFILE] structure of the process descriptor; this value is usually 1,024, but it can be raised if the process has root privileges.

The open_fds field initially contains the address of the open_fds_init field, which is a bitmap that identifies the file descriptors of currently opened files. The max_fdset field stores the number of bits in the bitmap. Because the fd_set data structure includes 1,024 bits, there is usually no need to expand the size of the bitmap. However, the kernel may dynamically expand the size of the bitmap if this turns out to be necessary, much as in the case of the array of file objects.

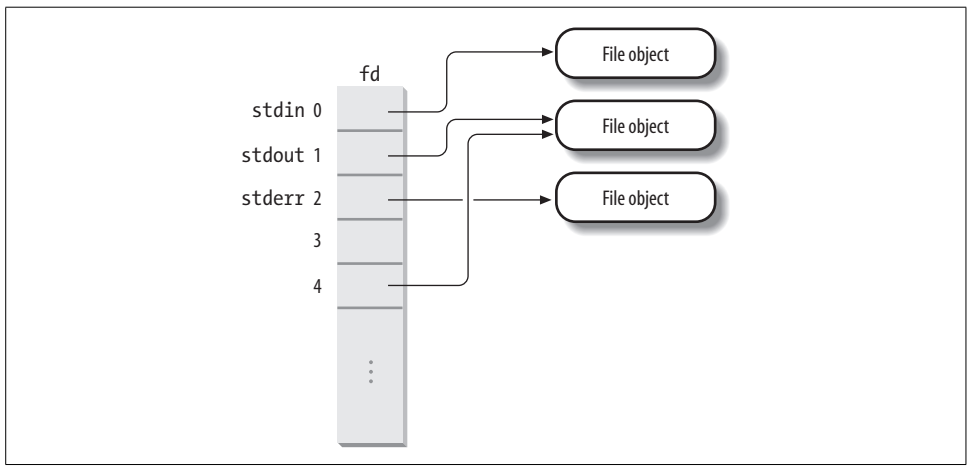


Figure 12-3. The fd array

The kernel provides an fget( ) function to be invoked when the kernel starts using a file object. This function receives as its parameter a file descriptor fd. It returns the address in current->files->fd[fd] (that is, the address of the corresponding file object), or NULL if no file corresponds to fd. In the first case, fget( ) increases the file object usage counter f_count by 1.

The kernel also provides an fput( ) function to be invoked when a kernel control path finishes using a file object. This function receives as its parameter the address of a file object and decreases its usage counter, f_count. Moreover, if this field becomes 0, the function invokes the release method of the file operations (if defined), decreases the i_writecount field in the inode object (if the file was opened for writing), removes the file object from the superblock's list, releases the file object to the slab allocator, and decreases the usage counters of the associated dentry object and of the filesystem descriptor (see the later section "Filesystem Mounting).

The fget_light( ) and fput_light( ) functions are faster versions of fget( ) and fput( ): the kernel uses them when it can safely assume that the current process already owns the file object—that is, the process has already previously increased the file object's reference counter. For instance, they are used by the service routines of the

system calls that receive a file descriptor as an argument, because the file object's reference counter has been increased by a previous open( ) system call.

Filesystem Types

The Linux kernel supports many different types of filesystems. In the following, we introduce a few special types of filesystems that play an important role in the internal design of the Linux kernel.

Next, we'll discuss filesystem registration—that is, the basic operation that must be performed, usually during system initialization, before using a filesystem type. Once a filesystem is registered, its specific functions are available to the kernel, so that type of filesystem can be mounted on the system's directory tree.

Special Filesystems

While network and disk-based filesystems enable the user to handle information stored outside the kernel, special filesystems may provide an easy way for system programs and administrators to manipulate the data structures of the kernel and to implement special features of the operating system. Table 12-8 lists the most common special filesystems used in Linux; for each of them, the table reports its suggested mount point and a short description.

Notice that a few filesystems have no fixed mount point (keyword "any" in the table). These filesystems can be freely mounted and used by the users. Moreover, some other special filesystems do not have a mount point at all (keyword "none" in the table). They are not for user interaction, but the kernel can use them to easily reuse some of the VFS layer code; for instance, we'll see in Chapter 19 that, thanks to the *pipefs* special filesystem, pipes can be treated in the same way as FIFO files.

Table 12-8. Most common special filesystems

Name	Mount point	Description
bdev	none	Block devices (see Chapter 13)
binfmt_misc	any	Miscellaneous executable formats (see Chapter 20)
devpts	*/dev/pts*	Pseudoterminal support (Open Group's Unix98 standard)
eventpollfs	none	Used by the efficient event polling mechanism
futexfs	none	Used by the futex (Fast Userspace Locking) mechanism
pipefs	none	Pipes (see Chapter 19)
proc	*/proc*	General access point to kernel data structures
rootfs	none	Provides an empty root directory for the bootstrap phase
shm	none	IPC-shared memory regions (see Chapter 19)
mqueue	any	Used to implement POSIX message queues (see Chapter 19)
sockfs	none	Sockets

Table 12-8. Most common special filesystems (continued)

Name	Mount point	Description
sysfs	*/sys*	General access point to system data (see Chapter 13)
tmpfs	any	Temporary files (kept in RAM unless swapped)
usbfs	*/proc/bus/usb*	USB devices

Special filesystems are not bound to physical block devices. However, the kernel assigns to each mounted special filesystem a fictitious block device that has the value 0 as major number and an arbitrary value (different for each special filesystem) as a minor number. The `set_anon_super()` function is used to initialize superblocks of special filesystems; this function essentially gets an unused minor number `dev` and sets the `s_dev` field of the new superblock with major number 0 and minor number `dev`. Another function called `kill_anon_super()` removes the superblock of a special filesystem. The `unnamed_dev_idr` variable includes pointers to auxiliary structures that record the minor numbers currently in use. Although some kernel designers dislike the fictitious block device identifiers, they help the kernel to handle special filesystems and regular ones in a uniform way.

We'll see a practical example of how the kernel defines and initializes a special filesystem in the later section "Mounting a Generic Filesystem."

Filesystem Type Registration

Often, the user configures Linux to recognize all the filesystems needed when compiling the kernel for his system. But the code for a filesystem actually may either be included in the kernel image or dynamically loaded as a module (see Appendix B). The VFS must keep track of all filesystem types whose code is currently included in the kernel. It does this by performing *filesystem type registration*.

Each registered filesystem is represented as a `file_system_type` object whose fields are illustrated in Table 12-9.

Table 12-9. The fields of the file_system_type object

Type	Field	Description
`const char *`	`name`	Filesystem name
`int`	`fs_flags`	Filesystem type flags
`struct super_block * (*)()`	`get_sb`	Method for reading a superblock
`void (*)()`	`kill_sb`	Method for removing a superblock
`struct module *`	`owner`	Pointer to the module implementing the filesystem (see Appendix B)
`struct file_system_type *`	`next`	Pointer to the next element in the list of filesystem types
`struct list_head`	`fs_supers`	Head of a list of superblock objects having the same filesystem type

All filesystem-type objects are inserted into a singly linked list. The file_systems variable points to the first item, while the next field of the structure points to the next item in the list. The file_systems_lock read/write spin lock protects the whole list against concurrent accesses.

The fs_supers field represents the head (first dummy element) of a list of superblock objects corresponding to mounted filesystems of the given type. The backward and forward links of a list element are stored in the s_instances field of the superblock object.

The get_sb field points to the filesystem-type-dependent function that allocates a new superblock object and initializes it (if necessary, by reading a disk). The kill_sb field points to the function that destroys a superblock.

The fs_flags field stores several flags, which are listed in Table 12-10.

Table 12-10. The filesystem type flags

Name	Description
FS_REQUIRES_DEV	Every filesystem of this type must be located on a physical disk device.
FS_BINARY_MOUNTDATA	The filesystem uses binary mount data.
FS_REVAL_DOT	Always revalidate the "." and ".." paths in the dentry cache (for network filesystems).
FS_ODD_RENAME	"Rename" operations are "move" operations (for network filesystems).

During system initialization, the register_filesystem( ) function is invoked for every filesystem specified at compile time; the function inserts the corresponding file_system_type object into the filesystem-type list.

The register_filesystem( ) function is also invoked when a module implementing a filesystem is loaded. In this case, the filesystem may also be unregistered (by invoking the unregister_filesystem( ) function) when the module is unloaded.

The get_fs_type( ) function, which receives a filesystem name as its parameter, scans the list of registered filesystems looking at the name field of their descriptors, and returns a pointer to the corresponding file_system_type object, if it is present.

Filesystem Handling

Like every traditional Unix system, Linux makes use of a *system's root filesystem*: it is the filesystem that is directly mounted by the kernel during the booting phase and that holds the system initialization scripts and the most essential system programs.

Other filesystems can be mounted—either by the initialization scripts or directly by the users—on directories of already mounted filesystems. Being a tree of directories, every filesystem has its own *root directory*. The directory on which a filesystem is mounted is called the *mount point*. A mounted filesystem is a *child* of the mounted

filesystem to which the mount point directory belongs. For instance, the */proc* virtual filesystem is a child of the system's root filesystem (and the system's root filesystem is the *parent* of */proc*). The root directory of a mounted filesystem hides the content of the mount point directory of the parent filesystem, as well as the whole subtree of the parent filesystem below the mount point.*

Namespaces

In a traditional Unix system, there is only one tree of mounted filesystems: starting from the system's root filesystem, each process can potentially access every file in a mounted filesystem by specifying the proper pathname. In this respect, Linux 2.6 is more refined: every process might have its own tree of mounted filesystems—the so-called *namespace* of the process.

Usually most processes share the same namespace, which is the tree of mounted filesystems that is rooted at the system's root filesystem and that is used by the *init* process. However, a process gets a new namespace if it is created by the `clone()` system call with the `CLONE_NEWNS` flag set (see the section "The clone(), fork(), and vfork() System Calls" in Chapter 3). The new namespace is then inherited by children processes if the parent creates them without the `CLONE_NEWNS` flag.

When a process mounts—or unmounts—a filesystem, it only modifies its namespace. Therefore, the change is visible to all processes that share the same namespace, and only to them. A process can even change the root filesystem of its namespace by using the Linux-specific `pivot_root()` system call.

The namespace of a process is represented by a `namespace` structure pointed to by the `namespace` field of the process descriptor. The fields of the `namespace` structure are shown in Table 12-11.

Table 12-11. The fields of the namespace structure

Type	Field	Description
`atomic_t`	`count`	Usage counter (how many processes share the namespace)
`struct vfsmount *`	`root`	Mounted filesystem descriptor for the root directory of the namespace
`struct list_head`	`list`	Head of list of all mounted filesystem descriptors
`struct rw_semaphore`	`sem`	Read/write semaphore protecting this structure

* The root directory of a filesystem can be different from the root directory of a process: as we have seen in the earlier section "Files Associated with a Process," the process's root directory is the directory corresponding to the "`/`" pathname. By default, the process' root directory coincides with the root directory of the system's root filesystem (or more precisely, with the root directory of the root filesystem in the namespace of the process, described in the following section), but it can be changed by invoking the `chroot()` system call.

The list field is the head of a doubly linked circular list collecting all mounted filesystems that belong to the namespace. The root field specifies the mounted filesystem that represents the root of the tree of mounted filesystems of this namespace. As we will see in the next section, mounted filesystems are represented by vfsmount structures.

Filesystem Mounting

In most traditional Unix-like kernels, each filesystem can be mounted only once. Suppose that an Ext2 filesystem stored in the */dev/fd0* floppy disk is mounted on */flp* by issuing the command:

```
mount -t ext2 /dev/fd0 /flp
```

Until the filesystem is unmounted by issuing a umount command, every other mount command acting on */dev/fd0* fails.

However, Linux is different: it is possible to mount the same filesystem several times. Of course, if a filesystem is mounted *n* times, its root directory can be accessed through *n* mount points, one per mount operation. Although the same filesystem can be accessed by using different mount points, it is really unique. Thus, there is only one superblock object for all of them, no matter of how many times it has been mounted.

Mounted filesystems form a hierarchy: the mount point of a filesystem might be a directory of a second filesystem, which in turn is already mounted over a third filesystem, and so on.*

It is also possible to stack multiple mounts on a single mount point. Each new mount on the same mount point hides the previously mounted filesystem, although processes already using the files and directories under the old mount can continue to do so. When the topmost mounting is removed, then the next lower mount is once more made visible.

As you can imagine, keeping track of mounted filesystems can quickly become a nightmare. For each mount operation, the kernel must save in memory the mount point and the mount flags, as well as the relationships between the filesystem to be mounted and the other mounted filesystems. Such information is stored in a *mounted filesystem descriptor* of type vfsmount. The fields of this descriptor are shown in Table 12-12.

* Quite surprisingly, the mount point of a filesystem might be a directory of the same filesystem, provided that it was already mounted. For instance:

```
mount -t ext2 /dev/fd0 /flp; touch /flp/foo
mkdir /flp/mnt; mount -t ext2 /dev/fd0 /flp/mnt
```

Now, the empty *foo* file on the floppy filesystem can be accessed both as */flp/foo* and */flp/mnt/foo*.

Table 12-12. The fields of the vfsmount data structure

Type	Field	Description
struct list_head	mnt_hash	Pointers for the hash table list.
struct vfsmount *	mnt_parent	Points to the parent filesystem on which this filesystem is mounted.
struct dentry *	mnt_mountpoint	Points to the dentry of the mount point directory where the filesystem is mounted.
struct dentry *	mnt_root	Points to the dentry of the root directory of this filesystem.
struct super_block *	mnt_sb	Points to the superblock object of this filesystem.
struct list_head	mnt_mounts	Head of a list including all filesystem descriptors mounted on directories of this filesystem.
struct list_head	mnt_child	Pointers for the mnt_mounts list of mounted filesystem descriptors.
atomic_t	mnt_count	Usage counter (increased to forbid filesystem unmounting).
int	mnt_flags	Flags.
int	mnt_expiry_mark	Flag set to true if the filesystem is marked as expired (the filesystem can be automatically unmounted if the flag is set and no one is using it).
char *	mnt_devname	Device filename.
struct list_head	mnt_list	Pointers for namespace's list of mounted filesystem descriptors.
struct list_head	mnt_fslink	Pointers for the filesystem-specific expire list.
struct namespace *	mnt_namespace	Pointer to the namespace of the process that mounted the filesystem.

The vfsmount data structures are kept in several doubly linked circular lists:

- A hash table indexed by the address of the vfsmount descriptor of the parent filesystem and the address of the dentry object of the mount point directory. The hash table is stored in the mount_hashtable array, whose size depends on the amount of RAM in the system. Each item of the table is the head of a circular doubly linked list storing all descriptors that have the same hash value. The mnt_hash field of the descriptor contains the pointers to adjacent elements in this list.
- For each namespace, a circular doubly linked list including all mounted filesystem descriptors belonging to the namespace. The list field of the namespace structure stores the head of the list, while the mnt_list field of the vfsmount descriptor contains the pointers to adjacent elements in the list.
- For each mounted filesystem, a circular doubly linked list including all child mounted filesystems. The head of each list is stored in the mnt_mounts field of the mounted filesystem descriptor; moreover, the mnt_child field of the descriptor stores the pointers to the adjacent elements in the list.

The vfsmount_lock spin lock protects the lists of mounted filesystem objects from concurrent accesses.

The mnt_flags field of the descriptor stores the value of several flags that specify how some kinds of files in the mounted filesystem are handled. These flags, which can be set through options of the *mount* command, are listed in Table 12-13.

Table 12-13. Mounted filesystem flags

Name	Description
MNT_NOSUID	Forbid setuid and setgid flags in the mounted filesystem
MNT_NODEV	Forbid access to device files in the mounted filesystem
MNT_NOEXEC	Disallow program execution in the mounted filesystem

Here are some functions that handle the mounted filesystem descriptors:

`alloc_vfsmnt(name)`
: Allocates and initializes a mounted filesystem descriptor

`free_vfsmnt(mnt)`
: Frees a mounted filesystem descriptor pointed to by mnt

`lookup_mnt(mnt, dentry)`
: Looks up a descriptor in the hash table and returns its address

Mounting a Generic Filesystem

We'll now describe the actions performed by the kernel in order to mount a filesystem. We'll start by considering a filesystem that is going to be mounted over a directory of an already mounted filesystem (in this discussion we will refer to this new filesystem as "generic").

The mount() system call is used to mount a generic filesystem; its sys_mount() service routine acts on the following parameters:

- The pathname of a device file containing the filesystem, or NULL if it is not required (for instance, when the filesystem to be mounted is network-based)
- The pathname of the directory on which the filesystem will be mounted (the mount point)
- The filesystem type, which must be the name of a registered filesystem
- The mount flags (permitted values are listed in Table 12-14)
- A pointer to a filesystem-dependent data structure (which may be NULL)

Table 12-14. Flags used by the mount() system call

Macro	Description
MS_RDONLY	Files can only be read
MS_NOSUID	Forbid setuid and setgid flags

Table 12-14. Flags used by the mount() system call (continued)

Macro	Description
MS_NODEV	Forbid access to device files
MS_NOEXEC	Disallow program execution
MS_SYNCHRONOUS	Write operations on files and directories are immediate
MS_REMOUNT	Remount the filesystem changing the mount flags
MS_MANDLOCK	Mandatory locking allowed
MS_DIRSYNC	Write operations on directories are immediate
MS_NOATIME	Do not update file access time
MS_NODIRATIME	Do not update directory access time
MS_BIND	Create a "bind mount," which allows making a file or directory visible at another point of the system directory tree (option --bind of the *mount* command)
MS_MOVE	Atomically move a mounted filesystem to another mount point (option --move of the *mount* command)
MS_REC	Recursively create "bind mounts" for a directory subtree
MS_VERBOSE	Generate kernel messages on mount errors

The sys_mount() function copies the value of the parameters into temporary kernel buffers, acquires the big kernel lock, and invokes the do_mount() function. Once do_mount() returns, the service routine releases the big kernel lock and frees the temporary kernel buffers.

The do_mount() function takes care of the actual mount operation by performing the following operations:

1. If some of the MS_NOSUID, MS_NODEV, or MS_NOEXEC mount flags are set, it clears them and sets the corresponding flag (MNT_NOSUID, MNT_NODEV, MNT_NOEXEC) in the mounted filesystem object.
2. Looks up the pathname of the mount point by invoking path_lookup(); this function stores the result of the pathname lookup in the local variable nd of type nameidata (see the later section "Pathname Lookup").
3. Examines the mount flags to determine what has to be done. In particular:
   a. If the MS_REMOUNT flag is specified, the purpose is usually to change the mount flags in the s_flags field of the superblock object and the mounted filesystem flags in the mnt_flags field of the mounted filesystem object. The do_remount() function performs these changes.
   b. Otherwise, it checks the MS_BIND flag. If it is specified, the user is asking to make visible a file or directory on another point of the system directory tree.
   c. Otherwise, it checks the MS_MOVE flag. If it is specified, the user is asking to change the mount point of an already mounted filesystem. The do_move_mount() function does this atomically.

d. Otherwise, it invokes do_new_mount(). This is the most common case. It is triggered when the user asks to mount either a special filesystem or a regular filesystem stored in a disk partition. do_new_mount() invokes the do_kern_mount() function passing to it the filesystem type, the mount flags, and the block device name. This function, which takes care of the actual mount operation and returns the address of a new mounted filesystem descriptor, is described below. Next, do_new_mount() invokes do_add_mount(), which essentially performs the following actions:

   1. Acquires for writing the namespace->sem semaphore of the current process, because the function is going to modify the namespace.
   2. The do_kern_mount() function might put the current process to sleep; meanwhile, another process might mount a filesystem on the very same mount point as ours or even change our root filesystem (current->namespace->root). Verifies that the lastly mounted filesystem on this mount point still refers to the current's namespace; if not, releases the read/write semaphore and returns an error code.
   3. If the filesystem to be mounted is already mounted on the mount point specified as parameter of the system call, or if the mount point is a symbolic link, it releases the read/write semaphore and returns an error code.
   4. Initializes the flags in the mnt_flags field of the new mounted filesystem object allocated by do_kern_mount().
   5. Invokes graft_tree() to insert the new mounted filesystem object in the namespace list, in the hash table, and in the children list of the parent-mounted filesystem.
   6. Releases the namespace->sem read/write semaphore and returns.

4. Invokes path_release() to terminate the pathname lookup of the mount point (see the later section "Pathname Lookup") and returns 0.

The do_kern_mount() function

The core of the mount operation is the do_kern_mount() function, which checks the filesystem type flags to determine how the mount operation is to be done. This function receives the following parameters:

fstype
: The name of the filesystem type to be mounted

flags
: The mount flags (see Table 12-14)

name
: The pathname of the block device storing the filesystem (or the filesystem type name for special filesystems)

`data`
: Pointer to additional data to be passed to the `read_super` method of the filesystem

The function takes care of the actual mount operation by performing essentially the following operations:

1. Invokes `get_fs_type()` to search in the list of filesystem types and locate the name stored in the `fstype` parameter; `get_fs_type()` returns in the local variable `type` the address of the corresponding `file_system_type` descriptor.
2. Invokes `alloc_vfsmnt()` to allocate a new mounted filesystem descriptor and stores its address in the `mnt` local variable.
3. Invokes the `type->get_sb()` filesystem-dependent function to allocate a new superblock and to initialize it (see below).
4. Initializes the `mnt->mnt_sb` field with the address of the new superblock object.
5. Initializes the `mnt->mnt_root` field with the address of the dentry object corresponding to the root directory of the filesystem, and increases the usage counter of the dentry object.
6. Initializes the `mnt->mnt_parent` field with the value in `mnt` (for generic filesystems, the proper value of `mnt_parent` will be set when the mounted filesystem descriptor is inserted in the proper lists by `graft_tree()`; see step 3d5 of `do_mount()`).
7. Initializes the `mnt->mnt_namespace` field with the value in `current->namespace`.
8. Releases the `s_umount` read/write semaphore of the superblock object (it was acquired when the object was allocated in step 3).
9. Returns the address `mnt` of the mounted filesystem object.

Allocating a superblock object

The `get_sb` method of the filesystem object is usually implemented by a one-line function. For instance, in the Ext2 filesystem the method is implemented as follows:

```
struct super_block * ext2_get_sb(struct file_system_type *type,
                        int flags, const char *dev_name, void *data)
{
    return get_sb_bdev(type, flags, dev_name, data, ext2_fill_super);
}
```

The `get_sb_bdev()` VFS function allocates and initializes a new superblock suitable for disk-based filesystems; it receives the address of the `ext2_fill_super()` function, which reads the disk superblock from the Ext2 disk partition.

To allocate superblocks suitable for special filesystems, the VFS also provides the `get_sb_pseudo()` function (for special filesystems with no mount point such as *pipefs*), the `get_sb_single()` function (for special filesystems with single mount point

such as *sysfs*), and the `get_sb_nodev()` function (for special filesystems that can be mounted several times such as *tmpfs*; see below).

The most important operations performed by `get_sb_bdev()` are the following:

1. Invokes `open_bdev_excl()` to open the block device having device file name `dev_name` (see the section "Character Device Drivers" in Chapter 13).
2. Invokes `sget()` to search the list of superblock objects of the filesystem (`type->fs_supers`, see the earlier section "Filesystem Type Registration"). If a superblock relative to the block device is already present, the function returns its address. Otherwise, it allocates and initializes a new superblock object, inserts it into the filesystem list and in the global list of superblocks, and returns its address.
3. If the superblock is not new (it was not allocated in the previous step, because the filesystem is already mounted), it jumps to step 6.
4. Copies the value of the `flags` parameter into the `s_flags` field of the superblock and sets the `s_id`, `s_old_blocksize`, and `s_blocksize` fields with the proper values for the block device.
5. Invokes the filesystem-dependent function passed as last argument to `get_sb_bdev()` to access the superblock information on disk and fill the other fields of the new superblock object.
6. Returns the address of the new superblock object.

Mounting the Root Filesystem

Mounting the root filesystem is a crucial part of system initialization. It is a fairly complex procedure, because the Linux kernel allows the root filesystem to be stored in many different places, such as a hard disk partition, a floppy disk, a remote filesystem shared via NFS, or even a *ramdisk* (a fictitious block device kept in RAM).

To keep the description simple, let's assume that the root filesystem is stored in a partition of a hard disk (the most common case, after all). While the system boots, the kernel finds the major number of the disk that contains the root filesystem in the `ROOT_DEV` variable (see Appendix A). The root filesystem can be specified as a device file in the */dev* directory either when compiling the kernel or by passing a suitable *"root"* option to the initial bootstrap loader. Similarly, the mount flags of the root filesystem are stored in the `root_mountflags` variable. The user specifies these flags either by using the *rdev* external program on a compiled kernel image or by passing a suitable *rootflags* option to the initial bootstrap loader (see Appendix A).

Mounting the root filesystem is a two-stage procedure, shown in the following list:

1. The kernel mounts the special *rootfs* filesystem, which simply provides an empty directory that serves as initial mount point.
2. The kernel mounts the real root filesystem over the empty directory.

Why does the kernel bother to mount the *rootfs* filesystem before the real one? Well, the *rootfs* filesystem allows the kernel to easily change the real root filesystem. In fact, in some cases, the kernel mounts and unmounts several root filesystems, one after the other. For instance, the initial bootstrap CD of a distribution might load in RAM a kernel with a minimal set of drivers, which mounts as root a minimal filesystem stored in a ramdisk. Next, the programs in this initial root filesystem probe the hardware of the system (for instance, they determine whether the hard disk is EIDE, SCSI, or whatever), load all needed kernel modules, and remount the root filesystem from a physical block device.

Phase 1: Mounting the rootfs filesystem

The first stage is performed by the `init_rootfs()` and `init_mount_tree()` functions, which are executed during system initialization.

The `init_rootfs()` function registers the special filesystem type *rootfs*:

```
struct file_system_type rootfs_fs_type = {
    .name = "rootfs";
    .get_sb = rootfs_get_sb;
    .kill_sb = kill_litter_super;
};
register_filesystem(&rootfs_fs_type);
```

The `init_mount_tree()` function executes the following operations:

1. Invokes `do_kern_mount()` passing to it the string `"rootfs"` as filesystem type, and stores the address of the mounted filesystem descriptor returned by this function in the `mnt` local variable. As explained in the previous section, `do_kern_mount()` ends up invoking the `get_sb` method of the *rootfs* filesystem, that is, the `rootfs_get_sb()` function:

   ```
   struct superblock *rootfs_get_sb(struct file_system_type *fs_type,
                           int flags, const char *dev_name, void *data)
   {
       return get_sb_nodev(fs_type, flags|MS_NOUSER, data,
                            ramfs_fill_super);
   }
   ```

   The `get_sb_nodev()` function, in turn, executes the following steps:

   a. Invokes `sget()` to allocate a new superblock passing as parameter the address of the `set_anon_super()` function (see the earlier section "Special Filesystems"). As a result, the `s_dev` field of the superblock is set in the appropriate way: major number 0, minor number different from those of other mounted special filesystems.

   b. Copies the value of the `flags` parameter into the `s_flags` field of the superblock.

   c. Invokes `ramfs_fill_super()` to allocate an inode object and a corresponding dentry object, and to fill the superblock fields. Because *rootfs* is a special

filesystem that has no disk superblock, only a couple of superblock operations need to be implemented.

d. Returns the address of the new superblock.

2. Allocates a `namespace` object for the namespace of process 0, and inserts into it the mounted filesystem descriptor returned by `do_kern_mount()`:

```
namespace = kmalloc(sizeof(*namespace), GFP_KERNEL);
list_add(&mnt->mnt_list, &namespace->list);
namespace->root = mnt;
mnt->mnt_namespace = init_task.namespace = namespace;
```

3. Sets the `namespace` field of every other process in the system to the address of the namespace object; also initializes the `namespace->count` usage counter. (By default, all processes share the same, initial namespace.)
4. Sets the root directory and the current working directory of process 0 to the root filesystem.

Phase 2: Mounting the real root filesystem

The second stage of the mount operation for the root filesystem is performed by the kernel near the end of the system initialization. There are several ways to mount the real root filesystem, according to the options selected when the kernel has been compiled and to the boot options passed by the kernel loader. For the sake of brevity, we consider the case of a disk-based filesystem whose device file name has been passed to the kernel by means of the "*root*" boot parameter. We also assume that no initial special filesystem is used, except the *rootfs* filesystem.

The `prepare_namespace()` function executes the following operations:

1. Sets the `root_device_name` variable with the device filename obtained from the "*root*" boot parameter. Also, sets the `ROOT_DEV` variable with the major and minor numbers of the same device file.
2. Invokes the `mount_root()` function, which in turn:
   a. Invokes `sys_mknod()` (the service routine of the `mknod()` system call) to create a */dev/root* device file in the *rootfs* initial root filesystem, having the major and minor numbers as in `ROOT_DEV`.
   b. Allocates a buffer and fills it with a list of filesystem type names. This list is either passed to the kernel in the "*rootfstype*" boot parameter or built by scanning the elements in the singly linked list of filesystem types.
   c. Scans the list of filesystem type names built in the previous step. For each name, it invokes `sys_mount()` to try to mount the given filesystem type on the root device. Because each filesystem-specific method uses a different magic number, all `get_sb()` invocations will fail except the one that attempts to fill the superblock by using the function of the filesystem really used on the root device. The filesystem is mounted on a directory named */root* of the *rootfs* filesystem.

d. Invokes `sys_chdir("/root")` to change the current directory of the process.

3. Moves the mount point of the mounted filesystem on the root directory of the *rootfs* filesystem:

```
sys_mount(".", "/", NULL, MS_MOVE, NULL);
sys_chroot(".");
```

Notice that the *rootfs* special filesystem is not unmounted: it is only hidden under the disk-based root filesystem.

Unmounting a Filesystem

The `umount()` system call is used to unmount a filesystem. The corresponding `sys_umount()` service routine acts on two parameters: a filename (either a mount point directory or a block device filename) and a set of flags. It performs the following actions:

1. Invokes `path_lookup()` to look up the mount point pathname; this function returns the results of the lookup operation in a local variable `nd` of type `nameidata` (see next section).
2. If the resulting directory is not the mount point of a filesystem, it sets the `retval` return code to `-EINVAL` and jumps to step 6. This check is done by verifying that `nd->mnt->mnt_root` contains the address of the dentry object pointed to by `nd.dentry`.
3. If the filesystem to be unmounted has not been mounted in the namespace, it sets the `retval` return code to `-EINVAL` and jumps to step 6. (Recall that some special filesystems have no mount point.) This check is done by invoking the `check_mnt()` function on `nd->mnt`.
4. If the user does not have the privileges required to unmount the filesystem, it sets the `retval` return code to `-EPERM` and jumps to step 6.
5. Invokes `do_umount()` passing as parameters `nd.mnt` (the mounted filesystem object) and `flags` (the set of flags). This function performs essentially the following operations:
   a. Retrieves the address of the `sb` superblock object from the `mnt_sb` field of the mounted filesystem object.
   b. If the user asked to force the unmount operation, it interrupts any ongoing mount operation by invoking the `umount_begin` superblock operation.
   c. If the filesystem to be unmounted is the root filesystem and the user didn't ask to actually detach it, it invokes `do_remount_sb()` to remount the root filesystem read-only and terminates.
   d. Acquires for writing the `namespace->sem` read/write semaphore of the current process, and gets the `vfsmount_lock` spin lock.
   e. If the mounted filesystem does not include mount points for any child mounted filesystem, or if the user asked to forcibly detach the filesystem, it

invokes `umount_tree()` to unmount the filesystem (together with all children filesystems).

f. Releases the `vfsmount_lock` spin lock and the `namespace->sem` read/write semaphore of the current process.

6. Decreases the usage counters of the dentry object corresponding to the root directory of the filesystem and of the mounted filesystem descriptor; these counters were increased by `path_lookup()`.
7. Returns the `retval` value.

Pathname Lookup

When a process must act on a file, it passes its file pathname to some VFS system call, such as `open()`, `mkdir()`, `rename()`, or `stat()`. In this section, we illustrate how the VFS performs a *pathname lookup*, that is, how it derives an inode from the corresponding file pathname.

The standard procedure for performing this task consists of analyzing the pathname and breaking it into a sequence of filenames. All filenames except the last must identify directories.

If the first character of the pathname is `/`, the pathname is absolute, and the search starts from the directory identified by `current->fs->root` (the process root directory). Otherwise, the pathname is relative, and the search starts from the directory identified by `current->fs->pwd` (the process-current directory).

Having in hand the dentry, and thus the inode, of the initial directory, the code examines the entry matching the first name to derive the corresponding inode. Then the directory file that has that inode is read from disk and the entry matching the second name is examined to derive the corresponding inode. This procedure is repeated for each name included in the path.

The dentry cache considerably speeds up the procedure, because it keeps the most recently used dentry objects in memory. As we saw before, each such object associates a filename in a specific directory to its corresponding inode. In many cases, therefore, the analysis of the pathname can avoid reading the intermediate directories from disk.

However, things are not as simple as they look, because the following Unix and VFS filesystem features must be taken into consideration:

- The access rights of each directory must be checked to verify whether the process is allowed to read the directory's content.
- A filename can be a symbolic link that corresponds to an arbitrary pathname; in this case, the analysis must be extended to all components of that pathname.

- Symbolic links may induce circular references; the kernel must take this possibility into account and break endless loops when they occur.
- A filename can be the mount point of a mounted filesystem. This situation must be detected, and the lookup operation must continue into the new filesystem.
- Pathname lookup has to be done inside the namespace of the process that issued the system call. The same pathname used by two processes with different namespaces may specify different files.

Pathname lookup is performed by the `path_lookup()` function, which receives three parameters:

`name`
: A pointer to the file pathname to be resolved.

`flags`
: The value of flags that represent how the looked-up file is going to be accessed. The allowed values are included later in Table 12-16.

`nd`
: The address of a `nameidata` data structure, which stores the results of the lookup operation and whose fields are shown in Table 12-15.

When `path_lookup()` returns, the `nameidata` structure pointed to by `nd` is filled with data pertaining to the pathname lookup operation.

Table 12-15. The fields of the nameidata data structure

Type	Field	Description
`struct dentry *`	`dentry`	Address of the dentry object
`struct vfs_mount *`	`mnt`	Address of the mounted filesystem object
`struct qstr`	`last`	Last component of the pathname (used when the `LOOKUP_PARENT` flag is set)
`unsigned int`	`flags`	Lookup flags
`int`	`last_type`	Type of last component of the pathname (used when the `LOOKUP_PARENT` flag is set)
`unsigned int`	`depth`	Current level of symbolic link nesting (see below); it must be smaller than 6
`char[ ] *`	`saved_names`	Array of pathnames associated with nested symbolic links
`union`	`intent`	One-member union specifying how the file will be accessed

The `dentry` and `mnt` fields point respectively to the dentry object and the mounted filesystem object of the last resolved component in the pathname. These two fields "describe" the file that is identified by the given pathname.

Because the dentry object and the mounted filesystem object returned by the `path_lookup()` function in the `nameidata` structure represent the result of a lookup operation, both objects should not be freed until the caller of `path_lookup()` finishes using

them. Therefore, path_lookup() increases the usage counters of both objects. If the caller wants to release these objects, it invokes the path_release() function passing as parameter the address of a nameidata structure.

The flags field stores the value of some flags used in the lookup operation; they are listed in Table 12-16. Most of these flags can be set by the caller in the flags parameter of path_lookup().

Table 12-16. The flags of the lookup operation

Macro	Description
LOOKUP_FOLLOW	If the last component is a symbolic link, interpret (follow) it
LOOKUP_DIRECTORY	The last component must be a directory
LOOKUP_CONTINUE	There are still filenames to be examined in the pathname
LOOKUP_PARENT	Look up the directory that includes the last component of the pathname
LOOKUP_NOALT	Do not consider the emulated root directory (useless in the 80x86 architecture)
LOOKUP_OPEN	Intent is to open a file
LOOKUP_CREATE	Intent is to create a file (if it doesn't exist)
LOOKUP_ACCESS	Intent is to check user's permission for a file

The path_lookup() function executes the following steps:

1. Initializes some fields of the nd parameter as follows:
   a. Sets the last_type field to LAST_ROOT (this is needed if the pathname is a slash or a sequence of slashes; see the later section "Parent Pathname Lookup").
   b. Sets the flags field to the value of the flags parameter
   c. Sets the depth field to 0.
2. Acquires for reading the current->fs->lock read/write semaphore of the current process.
3. If the first character in the pathname is a slash (/), the lookup operation must start from the root directory of current: the function gets the addresses of the corresponding mounted filesystem object (current->fs->rootmnt) and dentry object (current->fs->root), increases their usage counters, and stores the addresses in nd->mnt and nd->dentry, respectively.
4. Otherwise, if the first character in the pathname is not a slash, the lookup operation must start from the current working directory of current: the function gets the addresses of the corresponding mounted filesystem object (current->fs->pwdmnt) and dentry object (current->fs->pwd), increases their usage counters, and stores the addresses in nd->mnt and nd->dentry, respectively.
5. Releases the current->fs->lock read/write semaphore of the current process.
6. Sets the total_link_count field in the descriptor of the current process to 0 (see the later section "Lookup of Symbolic Links").

7. Invokes the link_path_walk() function to take care of the undergoing lookup operation:

```
return link_path_walk(name, nd);
```

We are now ready to describe the core of the pathname lookup operation, namely the link_path_walk() function. It receives as its parameters a pointer name to the pathname to be resolved and the address nd of a nameidata data structure.

To make things a bit easier, we first describe what link_path_walk() does when LOOKUP_PARENT is not set and the pathname does not contain symbolic links (standard pathname lookup). Next, we discuss the case in which LOOKUP_PARENT is set: this type of lookup is required when creating, deleting, or renaming a directory entry, that is, during a parent pathname lookup. Finally, we explain how the function resolves symbolic links.

Standard Pathname Lookup

When the LOOKUP_PARENT flag is cleared, link_path_walk() performs the following steps.

1. Initializes the lookup_flags local variable with nd->flags.
2. Skips all leading slashes (/) before the first component of the pathname.
3. If the remaining pathname is empty, it returns the value 0. In the nameidata data structure, the dentry and mnt fields point to the objects relative to the last resolved component of the original pathname.
4. If the depth field of the nd descriptor is positive, it sets the LOOKUP_FOLLOW flag in the lookup_flags local variable (see the section "Lookup of Symbolic Links").
5. Executes a cycle that breaks the pathname passed in the name parameter into components (the intermediate slashes are treated as filename separators); for each component found, the function:
   a. Retrieves the address of the inode object of the last resolved component from nd->dentry->d_inode. (In the first iteration, the inode refers to the directory from where to start the pathname lookup.)
   b. Checks that the permissions of the last resolved component stored into the inode allow execution (in Unix, a directory can be traversed only if it is executable). If the inode has a custom permission method, the function executes it; otherwise, it executes the exec_permission_lite() function, which examines the access mode stored in the i_mode inode field and the privileges of the running process. In both cases, if the last resolved component does not allow execution, link_path_walk() breaks out of the cycle and returns an error code.
   c. Considers the next component to be resolved. From its name, the function computes a 32-bit hash value to be used when looking in the dentry cache hash table.

d. Skips any trailing slash (/) after the slash that terminates the name of the component to be resolved.

e. If the component to be resolved is the last one in the original pathname, it jumps to step 6.

f. If the name of the component is "." (a single dot), it continues with the next component ("." refers to the current directory, so it has no effect inside a pathname).

g. If the name of the component is ".." (two dots), it tries to climb to the parent directory:

   1. If the last resolved directory is the process's root directory (`nd->dentry` is equal to `current->fs->root` and `nd->mnt` is equal to `current->fs->rootmnt`), then climbing is not allowed: it invokes `follow_mount()` on the last resolved component (see below) and continues with the next component.

   2. If the last resolved directory is the root directory of the `nd->mnt` filesystem (`nd->dentry` is equal to `nd->mnt->mnt_root`) and the `nd->mnt` filesystem is not mounted on top of another filesystem (`nd->mnt` is equal to `nd->mnt->mnt_parent`), then the `nd->mnt` filesystem is usually* the namespace's root filesystem: in this case, climbing is impossible, thus invokes `follow_mount()` on the last resolved component (see below) and continues with the next component.

   3. If the last resolved directory is the root directory of the `nd->mnt` filesystem and the `nd->mnt` filesystem is mounted on top of another filesystem, a filesystem switch is required. So, the function sets `nd->dentry` to `nd->mnt->mnt_mountpoint`, and `nd->mnt` to `nd->mnt->mnt_parent`, then restarts step 5g (recall that several filesystems can be mounted on the same mount point).

   4. If the last resolved directory is not the root directory of a mounted filesystem, then the function must simply climb to the parent directory: it sets `nd->dentry` to `nd->dentry->d_parent`, invokes `follow_mount()` on the parent directory, and continues with the next component.

   The `follow_mount()` function checks whether `nd->dentry` is a mount point for some filesystem (`nd->dentry->d_mounted` is greater than zero); in this case, it invokes `lookup_mnt()` to search the root directory of the mounted filesystem in the dentry cache, and updates `nd->dentry` and `nd->mnt` with the object addresses corresponding to the mounted filesystem; then, it repeats the whole operation (there can be several filesystems mounted on the same mount point). Essentially, invoking the `follow_mount()` function when

* This case can also occur for network filesystems disconnected from the namespace's directory tree.

climbing to the parent directory is required because the process could start the pathname lookup from a directory included in a filesystem hidden by another filesystem mounted over the parent directory.

h. The component name is neither "." nor "..", so the function must look it up in the dentry cache. If the low-level filesystem has a custom `d_hash` dentry method, the function invokes it to modify the hash value already computed in step 5c.

i. Sets the `LOOKUP_CONTINUE` flag in `nd->flags` to denote that there is a next component to be analyzed.

j. Invokes `do_lookup()` to derive the dentry object associated with a given parent directory (`nd->dentry`) and filename (the pathname component being resolved). The function essentially invokes `__d_lookup()` first to search the dentry object of the component in the dentry cache. If no such object exists, `do_lookup()` invokes `real_lookup()`. This latter function reads the directory from disk by executing the `lookup` method of the inode, creates a new dentry object and inserts it in the dentry cache, then creates a new inode object and inserts it into the inode cache.* At the end of this step, the `dentry` and `mnt` fields of the `next` local variable will point, respectively, to the dentry object and the mounted filesystem object of the component name to be resolved in this cycle.

k. Invokes the `follow_mount()` function to check whether the component just resolved (`next.dentry`) refers to a directory that is a mount point for some filesystem (`next.dentry->d_mounted` is greater than zero). `follow_mount()` updates `next.dentry` and `next.mnt` so that they point to the dentry object and mounted filesystem object of the upmost filesystem mounted on the directory specified by this pathname component (see step 5g).

l. Checks whether the component just resolved refers to a symbolic link (`next.dentry->d_inode` has a custom `follow_link` method). We'll deal with this case in the later section "Lookup of Symbolic Links."

m. Checks whether the component just resolved refers to a directory (`next.dentry->d_inode` has a custom `lookup` method). If not, returns the error `-ENOTDIR`, because the component is in the middle of the original pathname.

n. Sets `nd->dentry` to `next.dentry` and `nd->mnt` to `next.mnt`, then continues with the next component of the pathname.

6. Now all components of the original pathname are resolved except the last one. Clears the `LOOKUP_CONTINUE` flag in `nd->flags`.

* In a few cases, the function might find the required inode already in the inode cache. This happens when the pathname component is the last one and it does not refer to a directory, the corresponding file has several hard links, and finally the file has been recently accessed through a hard link different from the one used in this pathname.

7. If the pathname has a trailing slash, it sets the LOOKUP_FOLLOW and LOOKUP_DIRECTORY flags in the lookup_flags local variable to force the last component to be interpreted by later functions as a directory name.
8. Checks the value of the LOOKUP_PARENT flag in the lookup_flags variable. In the following, we assume that the flag is set to 0, and we postpone the opposite case to the next section.
9. If the name of the last component is "." (a single dot), terminates the execution and returns the value 0 (no error). In the nameidata structure that nd points to, the dentry and mnt fields refer to the objects relative to the next-to-last component of the pathname (each component "." has no effect inside a pathname).
10. If the name of the last component is ".." (two dots), it tries to climb to the parent directory:
    a. If the last resolved directory is the process's root directory (nd->dentry is equal to current->fs->root and nd->mnt is equal to current->fs->rootmnt), it invokes follow_mount() on the next-to-last component and terminates the execution and returns the value 0 (no error). nd->dentry and nd->mnt refer to the objects relative to the next-to-last component of the pathname—that is, to the root directory of the process.
    b. If the last resolved directory is the root directory of the nd->mnt filesystem (nd->dentry is equal to nd->mnt->mnt_root) and the nd->mnt filesystem is not mounted on top of another filesystem (nd->mnt is equal to nd->mnt->mnt_parent), then climbing is impossible, thus invokes follow_mount() on the next-to-last component and terminates the execution and returns the value 0 (no error).
    c. If the last resolved directory is the root directory of the nd->mnt filesystem and the nd->mnt filesystem is mounted on top of another filesystem, it sets nd->dentry to nd->mnt->mnt_mountpoint and nd->mnt to nd->mnt->mnt_parent, then restarts step 10.
    d. If the last resolved directory is not the root directory of a mounted filesystem, it sets nd->dentry to nd->dentry->d_parent, invokes follow_mount() on the parent directory, and terminates the execution and returns the value 0 (no error). nd->dentry and nd->mnt refer to the objects relative to the component preceding the next-to-last component of the pathname.
11. The name of the last component is neither "." nor "..", so the function must look it up in the dentry cache. If the low-level filesystem has a custom d_hash dentry method, the function invokes it to modify the hash value already computed in step 5c.
12. Invokes do_lookup() to derive the dentry object associated with the parent directory and the filename (see step 5j). At the end of this step, the next local variable contains the pointers to both the dentry and the mounted filesystem descriptor relative to the last component name.

13. Invokes follow_mount() to check whether the last component is a mount point for some filesystem and, if this is the case, to update the next local variable with the addresses of the dentry object and mounted filesystem object relative to the root directory of the upmost mounted filesystem.
14. Checks whether the LOOKUP_FOLLOW flag is set in lookup_flags and the inode object next.dentry->d_inode has a custom follow_link method. If this is the case, the component is a symbolic link that must be interpreted, as described in the later section "Lookup of Symbolic Links."
15. The component is not a symbolic link or the symbolic link should not be interpreted. Sets the nd->mnt and nd->dentry fields with the value stored in next.mnt and next.dentry, respectively. The final dentry object is the result of the whole lookup operation.
16. Checks whether nd->dentry->d_inode is NULL. This happens when there is no inode associated with the dentry object, usually because the pathname refers to a nonexistent file. In this case, the function returns the error code -ENOENT.
17. There is an inode associated with the last component of the pathname. If the LOOKUP_DIRECTORY flag is set in lookup_flags, it checks that the inode has a custom lookup method—that is, it is a directory. If not, the function returns the error code -ENOTDIR.
18. Returns the value 0 (no error). nd->dentry and nd->mnt refer to the last component of the pathname.

Parent Pathname Lookup

In many cases, the real target of a lookup operation is not the last component of the pathname, but the next-to-last one. For example, when a file is created, the last component denotes the filename of the not yet existing file, and the rest of the pathname specifies the directory in which the new link must be inserted. Therefore, the lookup operation should fetch the dentry object of the next-to-last component. For another example, unlinking a file identified by the pathname */foo/bar* consists of removing *bar* from the directory *foo*. Thus, the kernel is really interested in accessing the directory *foo* rather than *bar*.

The LOOKUP_PARENT flag is used whenever the lookup operation must resolve the directory containing the last component of the pathname, rather than the last component itself.

When the LOOKUP_PARENT flag is set, the link_path_walk() function also sets up the last and last_type fields of the nameidata data structure. The last field stores the name of the last component in the pathname. The last_type field identifies the type of the last component; it may be set to one of the values shown in Table 12-17.

Table 12-17. The values of the last_type field in the nameidata data structure

Value	Description
LAST_NORM	Last component is a regular filename
LAST_ROOT	Last component is "/" (that is, the entire pathname is "/")
LAST_DOT	Last component is "."
LAST_DOTDOT	Last component is ".."
LAST_BIND	Last component is a symbolic link into a special filesystem

The LAST_ROOT flag is the default value set by path_lookup() when the whole pathname lookup operation starts (see the description at the beginning of the section "Pathname Lookup"). If the pathname turns out to be simply "/", the kernel does not change the initial value of the last_type field.

The remaining values of the last_type field are set by link_path_walk() when the LOOKUP_PARENT flag is set; in this case, the function performs the same steps described in the previous section up to step 8. From step 8 onward, however, the lookup operation for the last component of the pathname is different:

1. Sets nd->last to the name of the last component.
2. Initializes nd->last_type to LAST_NORM.
3. If the name of the last component is "." (a single dot), it sets nd->last_type to LAST_DOT.
4. If the name of the last component is ".." (two dots), it sets nd->last_type to LAST_DOTDOT.
5. Returns the value 0 (no error).

As you can see, the last component is not interpreted at all. Thus, when the function terminates, the dentry and mnt fields of the nameidata data structure point to the objects relative to the directory that includes the last component.

Lookup of Symbolic Links

Recall that a symbolic link is a regular file that stores a pathname of another file. A pathname may include symbolic links, and they must be resolved by the kernel.

For example, if */foo/bar* is a symbolic link pointing to (containing the pathname) *../dir*, the pathname */foo/bar/file* must be resolved by the kernel as a reference to the file */dir/file*. In this example, the kernel must perform two different lookup operations. The first one resolves */foo/bar*: when the kernel discovers that *bar* is the name of a symbolic link, it must retrieve its content and interpret it as another pathname. The second pathname operation starts from the directory reached by the first operation and continues until the last component of the symbolic link pathname has been resolved. Next, the original lookup operation resumes from the dentry reached in the second one and with the component following the symbolic link in the original pathname.

To further complicate the scenario, the pathname included in a symbolic link may include other symbolic links. You might think that the kernel code that resolves the symbolic links is hard to understand, but this is not true; the code is actually quite simple because it is recursive.

However, untamed recursion is intrinsically dangerous. For instance, suppose that a symbolic link points to itself. Of course, resolving a pathname including such a symbolic link may induce an endless stream of recursive invocations, which in turn quickly leads to a kernel stack overflow. The `link_count` field in the descriptor of the current process is used to avoid the problem: the field is increased before each recursive execution and decreased right after. If a sixth nested lookup operation is attempted, the whole lookup operation terminates with an error code. Therefore, the level of nesting of symbolic links can be at most 5.

Furthermore, the `total_link_count` field in the descriptor of the current process keeps track of how many symbolic links (even nonnested) were followed in the original lookup operation. If this counter reaches the value 40, the lookup operation aborts. Without this counter, a malicious user could create a pathological pathname including many consecutive symbolic links that freeze the kernel in a very long lookup operation.

This is how the code basically works: once the `link_path_walk()` function retrieves the dentry object associated with a component of the pathname, it checks whether the corresponding inode object has a custom `follow_link` method (see step 5l and step 14 in the section "Standard Pathname Lookup"). If so, the inode is a symbolic link that must be interpreted before proceeding with the lookup operation of the original pathname.

In this case, the `link_path_walk()` function invokes `do_follow_link()`, passing to it the address `dentry` of the dentry object of the symbolic link and the address `nd` of the `nameidata` data structure. In turn, `do_follow_link()` performs the following steps:

1. Checks that `current->link_count` is less than 5; otherwise, it returns the error code `-ELOOP`.
2. Checks that `current->total_link_count` is less than 40; otherwise, it returns the error code `-ELOOP`.
3. Invokes `cond_resched()` to perform a process switch if required by the current process (flag `TIF_NEED_RESCHED` in the `thread_info` descriptor of the current process set).
4. Increases `current->link_count`, `current->total_link_count`, and `nd->depth`.
5. Updates the access time of the inode object associated with the symbolic link to be resolved.
6. Invokes the filesystem-dependent function that implements the `follow_link` method passing to it the `dentry` and `nd` parameters. This function extracts the pathname stored in the symbolic link's inode, and saves this pathname in the proper entry of the `nd->saved_names` array.

7. Invokes the `__vfs_follow_link()` function passing to it the address `nd` and the address of the pathname in the `nd->saved_names` array (see below).
8. If defined, executes the `put_link` method of the inode object, thus releasing the temporary data structures allocated by the `follow_link` method.
9. Decreases the `current->link_count` and `nd->depth` fields.
10. Returns the error code returned by the `__vfs_follow_link()` function (0 for no error).

In turn, the `__vfs_follow_link()` does essentially the following:

1. Checks whether the first character of the pathname stored in the symbolic link is a slash: in this case an absolute pathname has been found, so there is no need to keep in memory any information about the previous path. If so, invokes `path_release()` on the `nameidata` structure, thus releasing the objects resulting from the previous lookup steps; then, the function sets the `dentry` and `mnt` fields of the `nameidata` data structure to the current process root directory.
2. Invokes `link_path_walk()` to resolve the symbolic link pathname, passing to it as parameters the pathname and `nd`.
3. Returns the value taken from `link_path_walk()`.

When `do_follow_link()` finally terminates, it has set the `dentry` field of the `next` local variable with the address of the dentry object referred to by the symbolic link to the original execution of `link_path_walk()`. The `link_path_walk()` function can then proceed with the next step.

Implementations of VFS System Calls

For the sake of brevity, we cannot discuss the implementation of all the VFS system calls listed in Table 12-1. However, it could be useful to sketch out the implementation of a few system calls, in order to show how VFS's data structures interact.

Let's reconsider the example proposed at the beginning of this chapter: a user issues a shell command that copies the MS-DOS file */floppy/TEST* to the Ext2 file */tmp/test*. The command shell invokes an external program such as *cp*, which we assume executes the following code fragment:

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test", O_WRONLY | O_CREAT | O_TRUNC, 0600);
do {
    len = read(inf, buf, 4096);
    write(outf, buf, len);
} while (len);
close(outf);
close(inf);
```

Actually, the code of the real *cp* program is more complicated, because it must also check for possible error codes returned by each system call. In our example, we focus our attention on the "normal" behavior of a copy operation.

The open( ) System Call

The open( ) system call is serviced by the sys_open( ) function, which receives as its parameters the pathname filename of the file to be opened, some access mode flags flags, and a permission bit mask mode if the file must be created. If the system call succeeds, it returns a file descriptor—that is, the index assigned to the new file in the current->files->fd array of pointers to file objects; otherwise, it returns –1.

In our example, open( ) is invoked twice; the first time to open */floppy/TEST* for reading (O_RDONLY flag) and the second time to open */tmp/test* for writing (O_WRONLY flag). If */tmp/test* does not already exist, it is created (O_CREAT flag) with exclusive read and write access for the owner (octal 0600 number in the third parameter).

Conversely, if the file already exists, it is rewritten from scratch (O_TRUNC flag). Table 12-18 lists all flags of the open( ) system call.

Table 12-18. The flags of the open( ) system call

Flag name	Description
O_RDONLY	Open for reading
O_WRONLY	Open for writing
O_RDWR	Open for both reading and writing
O_CREAT	Create the file if it does not exist
O_EXCL	With O_CREAT, fail if the file already exists
O_NOCTTY	Never consider the file as a controlling terminal
O_TRUNC	Truncate the file (remove all existing contents)
O_APPEND	Always write at end of the file
O_NONBLOCK	No system calls will block on the file
O_NDELAY	Same as O_NONBLOCK
O_SYNC	Synchronous write (block until physical write terminates)
FASYNC	I/O event notification via signals
O_DIRECT	Direct I/O transfer (no kernel buffering)
O_LARGEFILE	Large file (size greater than 2 GB)
O_DIRECTORY	Fail if file is not a directory
O_NOFOLLOW	Do not follow a trailing symbolic link in pathname
O_NOATIME	Do not update the inode's last access time

Let's describe the operation of the sys_open( ) function. It performs the following steps:

1. Invokes getname( ) to read the file pathname from the process address space.
2. Invokes get_unused_fd( ) to find an empty slot in current->files->fd. The corresponding index (the new file descriptor) is stored in the fd local variable.
3. Invokes the filp_open( ) function, passing as parameters the pathname, the access mode flags, and the permission bit mask. This function, in turn, executes the following steps:
   a. Copies the access mode flags into namei_flags, but encodes the access mode flags O_RDONLY, O_WRONLY, and O_RDWR with a special format: the bit at index 0 (lowest-order) of namei_flags is set only if the file access requires read privileges; similarly, the bit at index 1 is set only if the file access requires write privileges. Notice that it is not possible to specify in the open( ) system call that a file access does not require either read or write privileges; this makes sense, however, in a pathname lookup operation involving symbolic links.
   b. Invokes open_namei( ), passing to it the pathname, the modified access mode flags, and the address of a local nameidata data structure. The function performs the lookup operation in the following manner:
      - If O_CREAT is not set in the access mode flags, starts the lookup operation with the LOOKUP_PARENT flag not set and the LOOKUP_OPEN flag set. Moreover, the LOOKUP_FOLLOW flag is set only if O_NOFOLLOW is cleared, while the LOOKUP_DIRECTORY flag is set only if the O_DIRECTORY flag is set.
      - If O_CREAT is set in the access mode flags, starts the lookup operation with the LOOKUP_PARENT, LOOKUP_OPEN, and LOOKUP_CREATE flags set. Once the path_lookup( ) function successfully returns, checks whether the requested file already exists. If not, allocates a new disk inode by invoking the create method of the parent inode.

      The open_namei( ) function also executes several security checks on the file located by the lookup operation. For instance, the function checks whether the inode associated with the dentry object found really exists, whether it is a regular file, and whether the current process is allowed to access it according to the access mode flags. Also, if the file is opened for writing, the function checks that the file is not locked by other processes.
   c. Invokes the dentry_open( ) function, passing to it the addresses of the dentry object and the mounted filesystem object located by the lookup operation, and the access mode flags. In turn, this function:
      1. Allocates a new file object.
      2. Initializes the f_flags and f_mode fields of the file object according to the access mode flags passed to the open( ) system call.
      3. Initializes the f_dentry and f_vfsmnt fields of the file object according to the addresses of the dentry object and the mounted filesystem object passed as parameters.

4. Sets the f_op field to the contents of the i_fop field of the corresponding inode object. This sets up all the methods for future file operations.
5. Inserts the file object into the list of opened files pointed to by the s_files field of the filesystem's superblock.
6. If the open method of the file operations is defined, the function invokes it.
7. Invokes file_ra_state_init() to initialize the read-ahead data structures (see Chapter 16).
8. If the O_DIRECT flag is set, it checks whether direct I/O operations can be performed on the file (see Chapter 16).
9. Returns the address of the file object.

d. Returns the address of the file object.

4. Sets current->files->fd[fd] to the address of the file object returned by dentry_open().
5. Returns fd.

The read( ) and write( ) System Calls

Let's return to the code in our *cp* example. The open( ) system calls return two file descriptors, which are stored in the inf and outf variables. Then the program starts a loop: at each iteration, a portion of the */floppy/TEST* file is copied into a local buffer (read( ) system call), and then the data in the local buffer is written into the */tmp/test* file (write( ) system call).

The read( ) and write( ) system calls are quite similar. Both require three parameters: a file descriptor fd, the address buf of a memory area (the buffer containing the data to be transferred), and a number count that specifies how many bytes should be transferred. Of course, read( ) transfers the data from the file into the buffer, while write( ) does the opposite. Both system calls return either the number of bytes that were successfully transferred or –1 to signal an error condition.

A return value less than count does not mean that an error occurred. The kernel is always allowed to terminate the system call even if not all requested bytes were transferred, and the user application must accordingly check the return value and reissue, if necessary, the system call. Typically, a small value is returned when reading from a pipe or a terminal device, when reading past the end of the file, or when the system call is interrupted by a signal. The end-of-file condition (EOF) can easily be recognized by a zero return value from read( ). This condition will not be confused with an abnormal termination due to a signal, because if read( ) is interrupted by a signal before a data is read, an error occurs.

The read or write operation always takes place at the file offset specified by the current file pointer (field f_pos of the file object). Both system calls update the file pointer by adding the number of transferred bytes to it.

In short, both sys_read( ) (the read( )'s service routine) and sys_write( ) (the write( )'s service routine) perform almost the same steps:

1. Invokes fget_light( ) to derive from fd the address file of the corresponding file object (see the earlier section "Files Associated with a Process").
2. If the flags in file->f_mode do not allow the requested access (read or write operation), it returns the error code -EBADF.
3. If the file object does not have a read( ) or aio_read( ) (write( ) or aio_write( )) file operation, it returns the error code -EINVAL.
4. Invokes access_ok( ) to perform a coarse check on the buf and count parameters (see the section "Verifying the Parameters" in Chapter 10).
5. Invokes rw_verify_area( ) to check whether there are conflicting mandatory locks for the file portion to be accessed. If so, it returns an error code, or puts the current process to sleep if the lock has been requested with a F_SETLKW command (see the section "File Locking" later in this chapter).
6. If defined, it invokes either the file->f_op->read or file->f_op->write method to transfer the data; otherwise, invokes either the file->f_op->aio_read or file->f_op->aio_write method. All these methods, which are discussed in Chapter 16, return the number of bytes that were actually transferred. As a side effect, the file pointer is properly updated.
7. Invokes fput_light( ) to release the file object.
8. Returns the number of bytes actually transferred.

The close( ) System Call

The loop in our example code terminates when the read( ) system call returns the value 0—that is, when all bytes of */floppy/TEST* have been copied into */tmp/test*. The program can then close the open files, because the copy operation has completed.

The close( ) system call receives as its parameter fd, which is the file descriptor of the file to be closed. The sys_close( ) service routine performs the following operations:

1. Gets the file object address stored in current->files->fd[fd]; if it is NULL, returns an error code.
2. Sets current->files->fd[fd] to NULL. Releases the file descriptor fd by clearing the corresponding bits in the open_fds and close_on_exec fields of current->files (see Chapter 20 for the Close on Execution flag).
3. Invokes filp_close( ), which performs the following operations:
   a. Invokes the flush method of the file operations, if defined.
   b. Releases all mandatory locks on the file, if any (see next section).
   c. Invokes fput( ) to release the file object.
4. Returns 0 or an error code. An error code can be raised by the flush method or by an error in a previous write operation on the file.

File Locking

When a file can be accessed by more than one process, a synchronization problem occurs. What happens if two processes try to write in the same file location? Or again, what happens if a process reads from a file location while another process is writing into it?

In traditional Unix systems, concurrent accesses to the same file location produce unpredictable results. However, Unix systems provide a mechanism that allows the processes to *lock* a file region so that concurrent accesses may be easily avoided.

The POSIX standard requires a file-locking mechanism based on the `fcntl()` system call. It is possible to lock an arbitrary region of a file (even a single byte) or to lock the whole file (including data appended in the future). Because a process can choose to lock only a part of a file, it can also hold multiple locks on different parts of the file.

This kind of lock does not keep out another process that is ignorant of locking. Like a semaphore used to protect a critical region in code, the lock is considered "advisory" because it doesn't work unless other processes cooperate in checking the existence of a lock before accessing the file. Therefore, POSIX's locks are known as *advisory locks*.

Traditional BSD variants implement advisory locking through the `flock()` system call. This call does not allow a process to lock a file region, only the whole file. Traditional System V variants provide the `lockf()` library function, which is simply an interface to `fcntl()`.

More importantly, System V Release 3 introduced *mandatory locking*: the kernel checks that every invocation of the `open()`, `read()`, and `write()` system calls does not violate a mandatory lock on the file being accessed. Therefore, mandatory locks are enforced even between noncooperative processes.*

Whether processes use advisory or mandatory locks, they can use both shared *read locks* and exclusive *write locks*. Several processes may have read locks on some file region, but only one process can have a write lock on it at the same time. Moreover, it is not possible to get a write lock when another process owns a read lock for the same file region, and vice versa.

* Oddly enough, a process may still unlink (delete) a file even if some other process owns a mandatory lock on it! This perplexing situation is possible because when a process deletes a file hard link, it does not modify its contents, but only the contents of its parent directory.

Linux File Locking

Linux supports all types of file locking: advisory and mandatory locks, plus the `fcntl()` and `flock()` system calls (`lockf()` is implemented as a standard library function).

The expected behavior of the `flock()` system call in every Unix-like operating system is to produce advisory locks only, without regard for the `MS_MANDLOCK` mount flag. In Linux, however, a special kind of `flock()`'s mandatory lock is used to support some proprietary network filesystems. It is the so-called *share-mode mandatory lock*; when set, no other process may open a file that would conflict with the access mode of the lock. Use of this feature for native Unix applications is discouraged, because the resulting source code will be nonportable.

Another kind of `fcntl()`-based mandatory lock called *lease* has been introduced in Linux. When a process tries to open a file protected by a lease, it is blocked as usual. However, the process that owns the lock receives a signal. Once informed, it should first update the file so that its content is consistent, and then release the lock. If the owner does not do this in a well-defined time interval (tunable by writing a number of seconds into */proc/sys/fs/lease-break-time*, usually 45 seconds), the lease is automatically removed by the kernel and the blocked process is allowed to continue.

A process can get or release an advisory file lock on a file in two possible ways:

- By issuing the `flock()` system call. The two parameters of the system call are the `fd` file descriptor, and a command to specify the lock operation. The lock applies to the whole file.
- By using the `fcntl()` system call. The three parameters of the system call are the `fd` file descriptor, a command to specify the lock operation, and a pointer to a `flock` structure (see Table 12-20). A couple of fields in this structure allow the process to specify the portion of the file to be locked. Processes can thus hold several locks on different portions of the same file.

Both the `fcntl()` and the `flock()` system call may be used on the same file at the same time, but a file locked through `fcntl()` does not appear locked to `flock()`, and vice versa. This has been done on purpose in order to avoid the deadlocks occurring when an application using a type of lock relies on a library that uses the other type.

Handling mandatory file locks is a bit more complex. Here are the steps to follow:

1. Mount the filesystem where mandatory locking is required using the `-o mand` option in the *mount* command, which sets the `MS_MANDLOCK` flag in the `mount()` system call. The default is to disable mandatory locking.
2. Mark the files as candidates for mandatory locking by setting their set-group bit (SGID) and clearing the group-execute permission bit. Because the set-group bit makes no sense when the group-execute bit is off, the kernel interprets that combination as a hint to use mandatory locks instead of advisory ones.
3. Uses the `fcntl()` system call (see below) to get or release a file lock.

Handling leases is much simpler than handling mandatory locks: it is sufficient to invoke a fcntl() system call with a F_SETLEASE or F_GETLEASE command. Another fcntl() invocation with the F_SETSIG command may be used to change the type of signal to be sent to the lease process holder.

Besides the checks in the read() and write() system calls, the kernel takes into consideration the existence of mandatory locks when servicing all system calls that could modify the contents of a file. For instance, an open() system call with the O_TRUNC flag set fails if any mandatory lock exists for the file.

The following section describes the main data structure used by the kernel to handle file locks issued by means of the flock() system call (FL_FLOCK locks) and of the fcntl() system call (FL_POSIX locks).

File-Locking Data Structures

All type of Linux locks are represented by the same file_lock data structure whose fields are shown in Table 12-19.

Table 12-19. The fields of the file_lock data structure

Type	Field	Description
struct file_lock *	fl_next	Next element in list of locks associated with the inode
struct list_head	fl_link	Pointers for active or blocked list
struct list_head	fl_block	Pointers for the lock's waiters list
struct files_struct *	fl_owner	Owner's files_struct
unsigned int	fl_pid	PID of the process owner
wait_queue_head_t	fl_wait	Wait queue of blocked processes
struct file *	fl_file	Pointer to file object
unsigned char	fl_flags	Lock flags
unsigned char	fl_type	Lock type
loff_t	fl_start	Starting offset of locked region
loff_t	fl_end	Ending offset of locked region
struct fasync_struct *	fl_fasync	Used for lease break notifications
unsigned long	fl_break_time	Remaining time before end of lease
struct file_lock_operations *	fl_ops	Pointer to file lock operations
struct lock_manager_operations *	fl_mops	Pointer to lock manager operations
union	fl_u	Filesystem-specific information

All lock_file structures that refer to the same file on disk are collected in a singly linked list, whose first element is pointed to by the i_flock field of the inode object. The fl_next field of the lock_file structure specifies the next element in the list.

When a process issues a blocking system call to require an exclusive lock while there are shared locks on the same file, the lock request cannot be satisfied immediately and the process must be suspended. The process is thus inserted into a wait queue pointed to by the `fl_wait` field of the blocked lock's `file_lock` structure. Two lists are used to distinguish lock requests that have been satisfied (*active locks*) from those that cannot be satisfied right away (*blocked locks*).

All active locks are linked together in the "global file lock list" whose head element is stored in the `file_lock_list` variable. Similarly, all blocked locks are linked together in the "blocked list" whose head element is stored in the `blocked_list` variable. The `fl_link` field is used to insert a `lock_file` structure in either one of these two lists.

Last but not least, the kernel must keep track of all blocked locks (the "waiters") associated with a given active lock (the "blocker"): this is the purpose of a list that links together all waiters with respect to a given blocker. The `fl_block` field of the blocker is the dummy head of the list, while the `fl_block` fields of the waiters store the pointers to the adjacent elements in the list.

FL_FLOCK Locks

An `FL_FLOCK` lock is always associated with a file object and is thus owned by the process that opened the file (or by all clone processes sharing the same opened file). When a lock is requested and granted, the kernel replaces every other lock that the process is holding on the same file object with the new lock. This happens only when a process wants to change an already owned read lock into a write one, or vice versa. Moreover, when a file object is being freed by the `fput( )` function, all `FL_FLOCK` locks that refer to the file object are destroyed. However, there could be other `FL_FLOCK` read locks set by other processes for the same file (inode), and they still remain active.

The `flock( )` system call allows a process to apply or remove an advisory lock on an open file. It acts on two parameters: the `fd` file descriptor of the file to be acted upon and a `cmd` parameter that specifies the lock operation. A `cmd` parameter of `LOCK_SH` requires a shared lock for reading, `LOCK_EX` requires an exclusive lock for writing, and `LOCK_UN` releases the lock.*

Usually this system call blocks the current process if the request cannot be immediately satisfied, for instance if the process requires an exclusive lock while some other process has already acquired the same lock. However, if the `LOCK_NB` flag is passed together with the `LOCK_SH` or `LOCK_EX` operation, the system call does not block; in other words, if the lock cannot be immediately obtained, the system call returns an error code.

* Actually, the `flock( )` system call can also establish share-mode mandatory locks by specifying the command `LOCK_MAND`. However, we'll not further discuss this case.

When the sys_flock( ) service routine is invoked, it performs the following steps:

1. Checks whether fd is a valid file descriptor; if not, returns an error code. Gets the address filp of the corresponding file object.
2. Checks that the process has read and/or write permission on the open file; if not, returns an error code.
3. Gets a new file_lock object lock and initializes it in the appropriate way: the fl_type field is set according to the value of the parameter cmd, the fl_file field is set to the address filp of the file object, the fl_flags field is set to FL_FLOCK, the fl_pid field is set to current->tgid, and the fl_end field is set to -1 to denote the fact that locking refers to the whole file (and not to a portion of it).
4. If the cmd parameter does not include the LOCK_NB bit, it adds to the fl_flags field the FL_SLEEP flag.
5. If the file has a flock file operation, the routine invokes it, passing as its parameters the file object pointer filp, a flag (F_SETLKW or F_SETLK depending on the value of the LOCK_NB bit), and the address of the new file_lock object lock.
6. Otherwise, if the flock file operation is not defined (the common case), invokes flock_lock_file_wait( ) to try to perform the required lock operation. Two parameters are passed: filp, a file object pointer, and lock, the address of the new file_lock object created in step 3.
7. If the file_lock descriptor has not been inserted in the active or blocked lists in the previous step, the routine releases it.
8. Returns 0 in case of success.

The flock_lock_file_wait( ) function executes a cycle consisting of the following steps:

1. Invokes flock_lock_file( ) passing as parameters the file object pointer filp and the address of the new file_lock object lock. This function performs, in turn, the following operations:
   a. Searches the list that filp->f_dentry->d_inode->i_flock points to. If an FL_FLOCK lock for the same file object is found, checks its type (LOCK_SH or LOCK_EX): if it is equal to the type of the new lock, returns 0 (nothing has to be done). Otherwise, the function removes the old element from the list of locks on the inode and the global file lock list, wakes up all processes sleeping in the wait queues of the locks in the fl_block list, and frees the file_lock structure.
   b. If the process is performing an unlock (LOCK_UN), nothing else needs to be done: the lock was nonexisting or it has already been released, thus returns 0.
   c. If an FL_FLOCK lock for the same file object has been found—thus the process is changing an already owned read lock into a write one (or vice versa)—gives some other higher-priority process, in particular every process previously blocked on the old file lock, a chance to run by invoking cond_resched( ).

d. Searches the list of locks on the inode again to verify that no existing FL_FLOCK lock conflicts with the requested one. There must be no FL_FLOCK write lock in the list, and moreover, there must be no FL_FLOCK lock at all if the process is requesting a write lock.

e. If no conflicting lock exists, it inserts the new file_lock structure into the inode's lock list and into the global file lock list, then returns 0 (success).

f. A conflicting lock has been found: if the FL_SLEEP flag in the fl_flags field is set, it inserts the new lock (the waiter lock) in the circular list of the blocker lock and in the global blocked list.

g. Returns the error code -EAGAIN.

2. Checks the return code of flock_lock_file( ):

   a. If the return code is 0 (no conflicting looks), it returns 0 (success).

   b. There are incompatibilities. If the FL_SLEEP flag in the fl_flags field is cleared, it releases the lock file_lock descriptor and returns -EAGAIN.

   c. Otherwise, there are incompatibilities but the process can sleep: invokes wait_event_interruptible( ) to insert the current process in the lock->fl_wait wait queue and to suspend it. When the process is awakened (right after the blocker lock has been released), it jumps to step 1 to retry the operation.

FL_POSIX Locks

An FL_POSIX lock is always associated with a process *and* with an inode; the lock is automatically released either when the process dies or when a file descriptor is closed (even if the process opened the same file twice or duplicated a file descriptor). Moreover, FL_POSIX locks are never inherited by a child across a fork( ).

When used to lock files, the fcntl( ) system call acts on three parameters: the fd file descriptor of the file to be acted upon, a cmd parameter that specifies the lock operation, and an fl pointer to a flock data structure* stored in the User Mode process address space; its fields are described in Table 12-20.

Table 12-20. The fields of the flock data structure

Type	Field	Description
short	l_type	F_RDLOCK (requests a shared lock), F_WRLOCK (requests an exclusive lock), F_UNLOCK (releases the lock)
short	l_whence	SEEK_SET (from beginning of file), SEEK_CURRENT (from current file pointer), SEEK_END (from end of file)

* Linux also defines a flock64 structure, which uses 64-bit long integers for the offset and length fields. In the following, we focus on the flock data structure, but the description is valid for flock64 too.

Table 12-20. The fields of the flock data structure (continued)

Type	Field	Description
off_t	l_start	Initial offset of the locked region relative to the value of l_whence
off_t	l_len	Length of locked region (0 means that the region includes all potential writes past the current end of the file)
pid_t	l_pid	PID of the owner

The sys_fcntl( ) service routine behaves differently, depending on the value of the flag set in the cmd parameter:

F_GETLK
: Determines whether the lock described by the flock structure conflicts with some FL_POSIX lock already obtained by another process. In this case, the flock structure is overwritten with the information about the existing lock.

F_SETLK
: Sets the lock described by the flock structure. If the lock cannot be acquired, the system call returns an error code.

F_SETLKW
: Sets the lock described by the flock structure. If the lock cannot be acquired, the system call blocks; that is, the calling process is put to sleep until the lock is available.

F_GETLK64, F_SETLK64, F_SETLKW64
: Identical to the previous ones, but the flock64 data structure is used rather than flock.

The sys_fcntl( ) service routine gets first a file object corresponding to the fd parameter and invokes then fcntl_getlk( ) or fcntl_setlk( ), depending on the command passed as its parameter (F_GETBLK for the former function, F_SETLK or F_SETLKW for the latter one). We'll consider the second case only.

The fcntl_setlk( ) function acts on three parameters: a filp pointer to the file object, a cmd command (F_SETLK or F_SETLKW), and a pointer to a flock data structure. The steps performed are the following:

1. Reads the structure pointed to by the fl parameter in a local variable of type flock.
2. Checks whether the lock should be a mandatory one and the file has a shared memory mapping (see the section "Memory Mapping" in Chapter 16). In this case, the function refuses to create the lock and returns the -EAGAIN error code, because the file is already being accessed by another process.
3. Initializes a new file_lock structure according to the contents of the user's flock structure and to the file size stored in the file's inode.
4. If the command is F_SETLKW, the function sets the FL_SLEEP flag in the fl_flags field of the file_lock structure.

5. If the l_type field in the flock structure is equal to F_RDLCK, it checks whether the process is allowed to read from the file; similarly, if l_type is equal to F_WRLCK, checks whether the process is allowed to write into the file. If not, it returns an error code.
6. Invokes the lock method of the file operations, if defined. Usually for disk-based filesystems, this method is not defined.
7. Invokes __posix_lock_file( ) passing as parameters the address of the file's inode object and the address of the file_lock object. This function performs, in turn, the following operations:
   a. Invokes posix_locks_conflict( ) for each FL_POSIX lock in the inode's lock list. The function checks whether the lock conflicts with the requested one. Essentially, there must be no FL_POSIX write lock for the same region in the inode list, and there may be no FL_POSIX lock at all for the same region if the process is requesting a write lock. However, locks owned by the same process never conflict; this allows a process to change the characteristics of a lock it already owns.
   b. If a conflicting lock is found, the function checks whether fcntl( ) was invoked with the F_SETLKW command. If so, the current process must be suspended: invokes posix_locks_deadlock( ) to check that no deadlock condition is being created among processes waiting for FL_POSIX locks, then inserts the new lock (waiter lock) both in the blocker list of the conflicting lock (blocker lock) and in the blocked list, and finally returns an error code. Otherwise, if fcntl( ) was invoked with the F_SETLK command, returns an error code.
   c. As soon as the inode's lock list includes no conflicting lock, the function checks all the FL_POSIX locks of the current process that overlap the file region that the current process wants to lock, and combines and splits adjacent areas as required. For example, if the process requested a write lock for a file region that falls inside a read-locked wider region, the previous read lock is split into two parts covering the nonoverlapping areas, while the central region is protected by the new write lock. In case of overlaps, newer locks always replace older ones.
   d. Inserts the new file_lock structure in the global file lock list and in the inode list.
   e. Returns the value 0 (success).
8. Checks the return code of __posix_lock_file( ):
   a. If the return code is 0 (no conflicting locks), it returns 0 (success).
   b. There are incompatibilities. If the FL_SLEEP flag in the fl_flags field is cleared, it releases the new file_lock descriptor and returns -EAGAIN.

c. Otherwise, if there are incompatibilities but the process can sleep, it invokes `wait_event_interruptible()` to insert the current process in the `lock->fl_wait` wait queue and to suspend it. When the process is awakened (right after the blocker lock has been released), it jumps to step 7 to retry the operation.