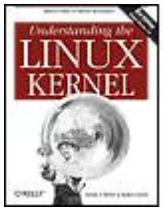


Understanding the Linux Kernel, 3rd Edition

By Daniel P. Bovet, Marco Cesati

.....

Publisher: **O'Reilly**
Pub Date: **November 2005**
ISBN: **0-596-00565-2**
Pages: **942**

Table of Contents | Index

Overview

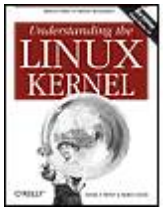
In order to thoroughly understand what makes Linux tick and why it works so well on a wide variety of systems, you need to delve deep into the heart of the kernel. The kernel handles all interactions between the CPU and the external world, and determines which programs will share processor time, in what order. It manages limited memory so well that hundreds of processes can share the system efficiently, and expertly organizes data transfers so that the CPU isn't kept waiting any longer than necessary for the relatively slow disks.

The third edition of Understanding the Linux Kernel takes you on a guided tour of the most significant data structures, algorithms, and programming tricks used in the kernel. Probing beyond superficial features, the authors offer valuable insights to people who want to know how things really work inside their machine. Important Intel-specific features are discussed. Relevant segments of code are dissected line by line. But the book covers more than just the functioning of the code; it explains the theoretical underpinnings of why Linux does things the way it does.

This edition of the book covers Version 2.6, which has seen significant changes to nearly every kernel subsystem, particularly in the areas of memory management and block devices. The book focuses on the following topics:

- Memory management, including file buffering, process swapping, and Direct memory Access (DMA)
- The Virtual Filesystem layer and the Second and Third Extended Filesystems
- Process creation and scheduling
- Signals, interrupts, and the essential interfaces to device drivers
- Timing
- Synchronization within the kernel
- Interprocess Communication (IPC)
- Program execution

Understanding the Linux Kernel will acquaint you with all the inner workings of Linux, but it's more than just an academic exercise. You'll learn what conditions bring out Linux's best performance, and you'll see how it meets the challenge of providing good system response during process scheduling, file access, and memory management in a wide variety of environments. This book will help you make the most of your Linux system.



Understanding the Linux Kernel, 3rd Edition
By Daniel P. Bovet, Marco Cesati
.....
Publisher: **O'Reilly**
Pub Date: **November 2005**
ISBN: **0-596-00565-2**
Pages: **942**

PREV NEXT

Understanding the Linux Kernel, Third Edition

by Daniel P. Bovet and Marco Cesati

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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

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

Printing History:

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

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The Linux series designations, Understanding the Linux Kernel, Third Edition, the image of a man with a bubble, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-00565-2

[M]

Preface

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

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

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

The Audience for This Book

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

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

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

Organization of the Material

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

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

-
- Efficient kernels take advantage of most available hardware features, such as addressing techniques, caches, processor exceptions, special instructions, processor control registers, and so on. If we want to convince you that the kernel indeed does quite a good job in performing a specific task, we must first tell what kind of support comes from the hardware.
-
- Even if a large portion of a Unix kernel source code is processor-independent and coded in C language, a small and critical part is coded in assembly language. A thorough knowledge of the kernel, therefore, requires the study of a few assembly language fragments that interact with the hardware.

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

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

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

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

Level of Description

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

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

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

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

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

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

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

Overview of the Book

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

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

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

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

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

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

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

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

Another issue to consider is disk access time; Chapter 15, The Page Cache, shows how a clever use of RAM reduces disk accesses, therefore improving system performance significantly. Building on the material covered in these last chapters, we can now explain in Chapter 16, Accessing Files, how user applications access normal files. Chapter 17, Page Frame Reclaiming, completes our discussion of Linux memory management and explains the techniques used by Linux to ensure that enough memory is always available. The last chapter dealing with files is Chapter 18, The Ext2 and Ext3 Filesystems, which illustrates the most frequently used Linux filesystem, namely Ext2 and its recent evolution, Ext3.

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

Last, but not least, are the appendixes: Appendix A, System Startup, sketches out how Linux is booted, while Appendix B, Modules, describes how to dynamically reconfigure the running kernel, adding and removing functionalities as needed. The Source Code Index includes all the Linux symbols referenced in

PREV NEXT

Background Information

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

PREV NEXT

PREV NEXT

Conventions in This Book

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

Constant Width

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

Italic

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

PREV NEXT

How to Contact Us

Please address comments and questions concerning this book to the publisher:
O'Reilly Media, Inc.1005 Gravenstein Highway NorthSebastopol, CA 95472(800) 998-9938 (in the United States or Canada)(707) 829-0515 (international or local)(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, or any additional information. You can access this page at:
http://www.oreilly.com/catalog/understandlk/

To comment or ask technical questions about this book, send email to:
bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:
http://www.oreilly.com

PREV NEXT

PREV NEXT

Safari? Enabled

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

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

PREV NEXT

Acknowledgments

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

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

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

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

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

Marco Cesati
July 2005

Daniel P. Bovet

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? is a registered trademark of Linus Torvalds.

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.

[†] 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.

1.1. 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.[*]

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

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 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

1.2. Hardware Dependency

Linux tries to maintain a neat distinction between hardware-dependent and hardware-independent source code. To that end, both the *arch* and the *include* directories include 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 MC680x0 microprocessors

1.3. 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 numbersfor instance, 2.6.10 and 2.6.11can 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.

1.4. 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 useras well as doing all the jobs the user bought the computer forbut 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.

1.4.1. Multiuser Systems

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

Multiuser operating systems must include several features:

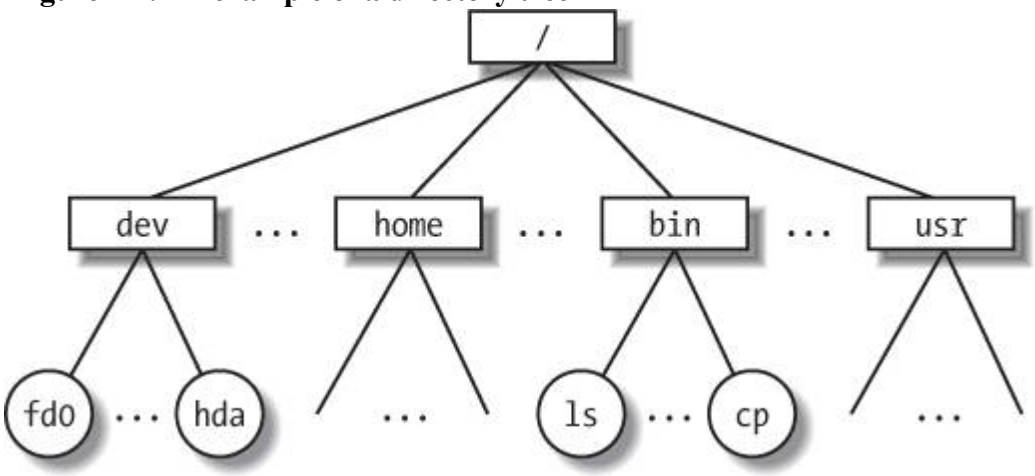
1.5. An Overview of the Unix Filesystem

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

1.5.1. Files

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

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.

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

Unix associates a current working directory with each process (see 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.

1.6. An Overview of Unix Kernels

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

1.6.1. The Process/Kernel Model

As already mentioned, a CPU can run in either User Mode or Kernel Mode . Actually, some CPUs can have more than two execution states. For instance, the 80 x 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

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 x 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.

2.1. Memory Addresses

Programmers casually refer to a memory address as the way to access the contents of a memory cell. But when dealing with 80 x 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 x 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.

2.2. 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).

2.2.1. 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.

2.2.2. Segment Descriptors

Each segment is represented by an 8-byte Segment Descriptor that describes the segment

2.3. Segmentation in Linux

Segmentation has been included in 80 x 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 x 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 232 -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

2.4. 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 x 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.

2.4.1. 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.[*]

[*] 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.

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 up to 220 entries (i.e., at 4

2.5. 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:

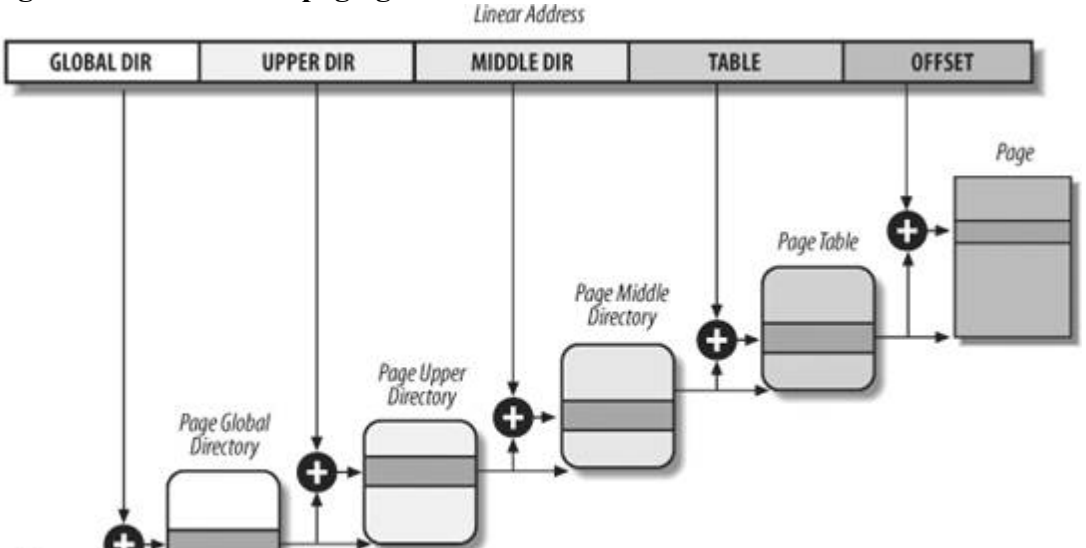
[*] This change has been made to fully support the linear address bit splitting used by the x86_64 platform (see Table 2-4).

- 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 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.

Figure 2-12. The Linux paging model



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).

3.1. 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 ."

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

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.

3.2.1. Process State

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

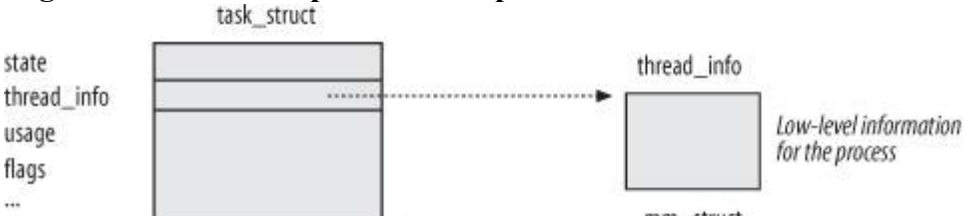
TASK_INTERRUPTIBLE

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 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.

Figure 3-1. The Linux process descriptor



3.3. 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.

3.3.1. Hardware Context

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

The set of data that must be loaded into the registers before the process resumes its execution on the CPU is called the hardware context . The hardware context is a subset of the process execution context, which includes all information needed for the process execution. In Linux, 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 80x86 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:

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

-
- Step-by-step switching performed through a sequence of mov instructions allows better control over the validity of the data being loaded. In particular, it is possible to check the values of 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.

3.3.2. Task State Segment

The 80x86 architecture includes a specific segment type called the Task State Segment (TSS), to store

3.4. 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.

3.4.1. 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.

3.5. 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).

3.5.1. 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.

3.5.1.1. 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

Chapter 4. Interrupts and Exceptions

An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside the CPU chip.

Interrupts are often divided into synchronous and asynchronous interrupts :

-
- Synchronous interrupts are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction.
-
- Asynchronous interrupts are generated by other hardware devices at arbitrary times with respect to the CPU clock signals.

Intel microprocessor manuals designate synchronous and asynchronous interrupts as exceptions and interrupts, respectively. We'll adopt this classification, although we'll occasionally use the term "interrupt signal" to designate both types together (synchronous as well as asynchronous).

Interrupts are issued by interval timers and I/O devices; for instance, the arrival of a keystroke from a user sets off an interrupt.

Exceptions, on the other hand, are caused either by programming errors or by anomalous conditions that must be handled by the kernel. In the first case, the kernel handles the exception by delivering to the current process one of the signals familiar to every Unix programmer. In the second case, the kernel performs all the steps needed to recover from the anomalous condition, such as a Page Fault or a requestvia an assembly language instruction such as int or sysenter for a kernel service.

We start by describing in the next section the motivation for introducing such signals. We then show how the well-known IRQs (Interrupt ReQuests) issued by I/O devices give rise to interrupts, and we detail how 80 x 86 processors handle interrupts and exceptions at the hardware level. Then we illustrate, in the section "Initializing the Interrupt Descriptor Table," how Linux initializes all the data structures required by the 80x86 interrupt architecture. The remaining three sections describe how Linux handles interrupt signals at the software level.

One word of caution before moving on: in this chapter, we cover only "classic" interrupts common to all PCs; we do not cover the nonstandard interrupts of some architectures.

4.1. The Role of Interrupt Signals

As the name suggests, interrupt signals provide a way to divert the processor to code outside the normal flow of control. When an interrupt signal arrives, the CPU must stop what it's currently doing and switch to a new activity; it does this by saving the current value of the program counter (i.e., the content of the eip and cs registers) in the Kernel Mode stack and by placing an address related to the interrupt type into the program counter.

There are some things in this chapter that will remind you of the context switch described in the previous chapter, carried out when a kernel substitutes one process for another. But there is a key difference between interrupt handling and process switching: the code executed by an interrupt or by an exception handler is not a process. Rather, it is a kernel control path that runs at the expense of the same process that was running when the interrupt occurred (see the later section "Nested Execution of Exception and Interrupt Handlers"). As a kernel control path, the interrupt handler is lighter than a process (it has less context and requires less time to set up or tear down).

Interrupt handling is one of the most sensitive tasks performed by the kernel, because it must satisfy the following constraints:

-
- Interrupts can come anytime, when the kernel may want to finish something else it was trying to do. The kernel's goal is therefore to get the interrupt out of the way as soon as possible and defer as much processing as it can. For instance, suppose a block of data has arrived on a network line. When the hardware interrupts the kernel, it could simply mark the presence of data, give the processor back to whatever was running before, and do the rest of the processing later (such as moving the data into a buffer where its recipient process can find it, and then restarting the process). The activities that the kernel needs to perform in response to an interrupt are thus divided into a critical urgent part that the kernel executes right away and a deferrable part that is left for later.
-
- Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs. This should be allowed as much as possible, because it keeps the I/O devices busy (see the later section "Nested Execution of Exception and Interrupt Handlers"). As a result, the interrupt handlers must be coded so that the corresponding kernel control paths can be executed in a nested manner. When the last kernel control path terminates, the kernel must be able to resume execution of the interrupted process or switch to another process if the interrupt signal has caused a rescheduling activity.
-
- Although the kernel may accept a new interrupt signal while handling a previous one, some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible because, according to the previous requirement, the kernel, and particularly the interrupt handlers, should run most of the time with the interrupts enabled.

4.2. Interrupts and Exceptions

The Intel documentation classifies interrupts and exceptions as follows:

-
- Interrupts:

- Maskable interrupts
- All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts . A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.

- Nonmaskable interrupts
- Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts . Nonmaskable interrupts are always recognized by the CPU.
-
- Exceptions:

- Processor-detected exceptions
- Generated when the CPU detects an anomalous condition while executing an instruction. These are further divided into three groups, depending on the value of the eip register that is saved on the Kernel Mode stack when the CPU control unit raises the exception.

- Faults
- Can generally be corrected; once corrected, the program is allowed to restart with no loss of continuity. The saved value of eip is the address of the instruction that caused the fault, and hence that instruction can be resumed when the exception handler terminates. As we'll see in the section "Page Fault Exception Handler" in Chapter 9, resuming the same instruction is necessary whenever the handler is able to correct the anomalous condition that caused the exception.

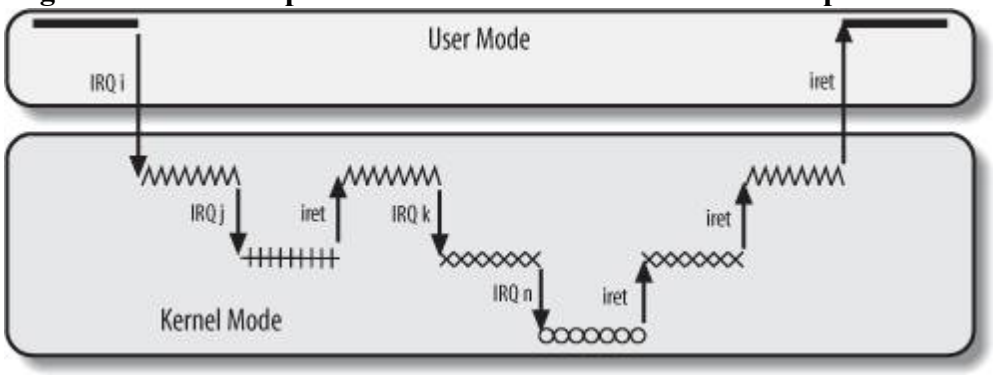
- Traps
- Reported immediately following the execution of the trapping instruction; after the kernel returns control to the program, it is allowed to continue its execution with no loss of continuity. The saved value of eip is the address of the instruction that should be executed after the one that caused the trap. A trap is triggered only when there is no need to reexecute the instruction that terminated. The main use of traps is for debugging purposes. The role of the interrupt signal in this case is to notify the debugger that a specific instruction has been executed (for instance, a breakpoint has been reached within a program). Once the user has examined the data provided by the debugger, she may ask that execution of the debugged program resume, starting from the next instruction.

4.3. Nested Execution of Exception and Interrupt Handlers

Every interrupt or exception gives rise to a kernel control path or separate sequence of instructions that execute in Kernel Mode on behalf of the current process. For instance, when an I/O device raises an interrupt, the first instructions of the corresponding kernel control path are those that save the contents of the CPU registers in the Kernel Mode stack, while the last are those that restore the contents of the registers.

Kernel control paths may be arbitrarily nested; an interrupt handler may be interrupted by another interrupt handler, thus giving rise to a nested execution of kernel control paths , as shown in Figure 4-3. As a result, the last instructions of a kernel control path that is taking care of an interrupt do not always put the current process back into User Mode: if the level of nesting is greater than 1, these instructions will put into execution the kernel control path that was interrupted last, and the CPU will continue to run in Kernel Mode.

Figure 4-3. An example of nested execution of kernel control paths



The price to pay for allowing nested kernel control paths is that an interrupt handler must never block, that is, no process switch can take place until an interrupt handler is running. In fact, all the data needed to resume a nested kernel control path is stored in the Kernel Mode stack, which is tightly bound to the current process.

Assuming that the kernel is bug free, most exceptions can occur only while the CPU is in User Mode. Indeed, they are either caused by programming errors or triggered by debuggers. However, the "Page Fault " exception may occur in Kernel Mode. This happens when the process attempts to address a page that belongs to its address space but is not currently in RAM. While handling such an exception, the kernel may suspend the current process and replace it with another one until the requested page is available. The kernel control path that handles the "Page Fault" exception resumes execution as soon as the process gets the processor again.

Because the "Page Fault" exception handler never gives rise to further exceptions, at most two kernel control paths associated with exceptions (the first one caused by a system call invocation, the second one caused by a Page Fault) may be stacked, one on top of the other.

In contrast to exceptions, interrupts issued by I/O devices do not refer to data structures specific to the current process, although the kernel control paths that handle them run on behalf of that process. As a matter of fact, it is impossible to predict which process will be running when a given interrupt occurs.

An interrupt handler may preempt both other interrupt handlers and exception handlers. Conversely, an exception handler never preempts an interrupt handler. The only exception that can be triggered in Kernel Mode is "Page Fault," which we just described. But interrupt handlers never perform operations that can induce page faults, and thus, potentially, a process switch.

4.4. Initializing the Interrupt Descriptor Table

Now that we understand what the 80x86 microprocessors do with interrupts and exceptions at the hardware level, we can move on to describe how the Interrupt Descriptor Table is initialized.

Remember that before the kernel enables the interrupts, it must load the initial address of the IDT table into the idtr register and initialize all the entries of that table. This activity is done while initializing the system (see Appendix A).

The int instruction allows a User Mode process to issue an interrupt signal that has an arbitrary vector ranging from 0 to 255. Therefore, initialization of the IDT must be done carefully, to block illegal interrupts and exceptions simulated by User Mode processes via int instructions. This can be achieved by setting the DPL field of the particular Interrupt or Trap Gate Descriptor to 0. If the process attempts to issue one of these interrupt signals, the control unit checks the CPL value against the DPL field and issues a "General protection " exception.

In a few cases, however, a User Mode process must be able to issue a programmed exception. To allow this, it is sufficient to set the DPL field of the corresponding Interrupt or Trap Gate Descriptors to 3 that is, as high as possible.

Let's now see how Linux implements this strategy.

4.4.1. Interrupt, Trap, and System Gates

As mentioned in the earlier section "Interrupt Descriptor Table," Intel provides three types of interrupt descriptors : Task, Interrupt, and Trap Gate Descriptors. Linux uses a slightly different breakdown and terminology from Intel when classifying the interrupt descriptors included in the Interrupt Descriptor Table:

Interrupt gate

An Intel interrupt gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). All Linux interrupt handlers are activated by means of interrupt gates , and all are restricted to Kernel Mode.

System gate

An Intel trap gate that can be accessed by a User Mode process (the gate's DPL field is equal to 3). The three Linux exception handlers associated with the vectors 4, 5, and 128 are activated by means of system gates , so the three assembly language instructions into , bound , and int $0x80 can be issued in User Mode.

System interrupt gate

An Intel interrupt gate that can be accessed by a User Mode process (the gate's DPL field is equal to 3). The exception handler associated with the vector 3 is activated by means of a system interrupt gate, so the assembly language instruction int3 can be issued in User Mode.

Trap gate

4.5. Exception Handling

Most exceptions issued by the CPU are interpreted by Linux as error conditions. When one of them occurs, the kernel sends a signal to the process that caused the exception to notify it of an anomalous condition. If, for instance, a process performs a division by zero, the CPU raises a "Divide error " exception, and the corresponding exception handler sends a SIGFPE signal to the current process, which then takes the necessary steps to recover or (if no signal handler is set for that signal) abort.

There are a couple of cases, however, where Linux exploits CPU exceptions to manage hardware resources more efficiently. A first case is already described in the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3. The "Device not available " exception is used together with the TS flag of the cr0 register to force the kernel to load the floating point registers of the CPU with new values. A second case involves the "Page Fault " exception, which is used to defer allocating new page frames to the process until the last possible moment. The corresponding handler is complex because the exception may, or may not, denote an error condition (see the section "Page Fault Exception Handler" in Chapter 9).

Exception handlers have a standard structure consisting of three steps:

1.
1. Save the contents of most registers in the Kernel Mode stack (this part is coded in assembly language).
2.
2. Handle the exception by means of a high-level C function.
3.
3. Exit from the handler by means of the ret_from_exception( ) function.

To take advantage of exceptions, the IDT must be properly initialized with an exception handler function for each recognized exception. It is the job of the trap_init( ) function to insert the final valuesthe functions that handle the exceptionsinto all IDT entries that refer to nonmaskable interrupts and exceptions. This is accomplished through the set_trap_gate( ), set_intr_gate( ), set_system_gate( ), set_system_intr_gate( ), and set_task_gate( ) functions:

```
set_trap_gate(0,&divide_error);
set_trap_gate(1,&debug);
set_intr_gate(2,&nmi);
set_system_intr_gate(3,&int3);
set_system_gate(4,&overflow);
set_system_gate(5,&bounds);
set_trap_gate(6,&invalid_op);
set_trap_gate(7,&device_not_available);
set_task_gate(8,31);
set_trap_gate(9,&coprocessor_segment_overrun);
set_trap_gate(10,&invalid_TSS);
set_trap_gate(11,&segment_not_present);
set_trap_gate(12,&stack_segment);
set_trap_gate(13,&general_protection);
set_intr_gate(14,&page_fault);
set_trap_gate(16,&coprocessor_error);
set_trap_gate(17,&alignment_check);
set_trap_gate(18,&machine_check);
set_trap_gate(19,&simd_coprocessor_error);
set_system_gate(128,&system_call);
```


The "Double fault" exception is handled by means of a task gate instead of a trap or system gate, because

4.6. Interrupt Handling

As we explained earlier, most exceptions are handled simply by sending a Unix signal to the process that caused the exception. The action to be taken is thus deferred until the process receives the signal; as a result, the kernel is able to process the exception quickly.

This approach does not hold for interrupts, because they frequently arrive long after the process to which they are related (for instance, a process that requested a data transfer) has been suspended and a completely unrelated process is running. So it would make no sense to send a Unix signal to the current process.

Interrupt handling depends on the type of interrupt. For our purposes, we'll distinguish three main classes of interrupts:

I/O interrupts

An I/O device requires attention; the corresponding interrupt handler must query the device to determine the proper course of action. We cover this type of interrupt in the later section "I/O Interrupt Handling."

Timer interrupts

Some timer, either a local APIC timer or an external timer, has issued an interrupt; this kind of interrupt tells the kernel that a fixed-time interval has elapsed. These interrupts are handled mostly as I/O interrupts; we discuss the peculiar characteristics of timer interrupts in Chapter 6.

Interprocessor interrupts

A CPU issued an interrupt to another CPU of a multiprocessor system. We cover such interrupts in the later section "Interprocessor Interrupt Handling."

4.6.1. I/O Interrupt Handling

In general, an I/O interrupt handler must be flexible enough to service several devices at the same time. In the PCI bus architecture, for instance, several devices may share the same IRQ line. This means that the interrupt vector alone does not tell the whole story. In the example shown in Table 4-3, the same vector 43 is assigned to the USB port and to the sound card. However, some hardware devices found in older PC architectures (such as ISA) do not reliably operate if their IRQ line is shared with other devices.

Interrupt handler flexibility is achieved in two distinct ways, as discussed in the following list.

IRQ sharing

The interrupt handler executes several interrupt service routines (ISRs). Each ISR is a function related to a single device sharing the IRQ line. Because it is not possible to know in advance which particular device issued the IRQ, each ISR is executed to verify whether its device needs attention; if so, the ISR performs all the operations that need to be executed when the device raises an interrupt.

IRQ dynamic allocation

4.7. Softirqs and Tasklets

We mentioned earlier in the section "Interrupt Handling" that several tasks among those executed by the kernel are not critical: they can be deferred for a long period of time, if necessary. Remember that the interrupt service routines of an interrupt handler are serialized, and often there should be no occurrence of an interrupt until the corresponding interrupt handler has terminated. Conversely, the deferrable tasks can execute with all interrupts enabled. Taking them out of the interrupt handler helps keep kernel response time small. This is a very important property for many time-critical applications that expect their interrupt requests to be serviced in a few milliseconds.

Linux 2.6 answers such a challenge by using two kinds of non-urgent interruptible kernel functions: the so-called deferrable functions[*] (softirqs and tasklets), and those executed by means of some work queues (we will describe them in the section "Work Queues" later in this chapter).

[*] These are also called software interrupts, but we denote them as "deferrable functions" to avoid confusion with programmed exceptions, which are referred to as "software interrupts " in Intel manuals.

Softirqs and tasklets are strictly correlated, because tasklets are implemented on top of softirqs. As a matter of fact, the term "softirq," which appears in the kernel source code, often denotes both kinds of deferrable functions. Another widely used term is interrupt context : it specifies that the kernel is currently executing either an interrupt handler or a deferrable function.

Softirqs are statically allocated (i.e., defined at compile time), while tasklets can also be allocated and initialized at runtime (for instance, when loading a kernel module). Softirqs can run concurrently on several CPUs, even if they are of the same type. Thus, softirqs are reentrant functions and must explicitly protect their data structures with spin locks. Tasklets do not have to worry about this, because their execution is controlled more strictly by the kernel. Tasklets of the same type are always serialized: in other words, the same type of tasklet cannot be executed by two CPUs at the same time. However, tasklets of different types can be executed concurrently on several CPUs. Serializing the tasklet simplifies the life of device driver developers, because the tasklet function needs not be reentrant.

Generally speaking, four kinds of operations can be performed on deferrable functions:

Initialization

Defines a new deferrable function; this operation is usually done when the kernel initializes itself or a module is loaded.

Activation

Marks a deferrable function as "pending" to be run the next time the kernel schedules a round of executions of deferrable functions. Activation can be done at any time (even while handling interrupts).

Masking

Selectively disables a deferrable function so that it will not be executed by the kernel even if activated. We'll see in the section "Disabling and Enabling Deferrable Functions" in Chapter 5 that disabling deferrable functions is sometimes essential.

Execution

4.8. Work Queues

The work queues have been introduced in Linux 2.6 and replace a similar construct called "task queue" used in Linux 2.4. They allow kernel functions to be activated (much like deferrable functions) and later executed by special kernel threads called worker threads .

Despite their similarities, deferrable functions and work queues are quite different. The main difference is that deferrable functions run in interrupt context while functions in work queues run in process context. Running in process context is the only way to execute functions that can block (for instance, functions that need to access some block of data on disk) because, as already observed in the section "Nested Execution of Exception and Interrupt Handlers" earlier in this chapter, no process switch can take place in interrupt context. Neither deferrable functions nor functions in a work queue can access the User Mode address space of a process. In fact, a deferrable function cannot make any assumption about the process that is currently running when it is executed. On the other hand, a function in a work queue is executed by a kernel thread, so there is no User Mode address space to access.

4.8.1.

4.8.1.1. Work queue data structures

The main data structure associated with a work queue is a descriptor called workqueue_struct, which contains, among other things, an array of NR_CPUS elements, the maximum number of CPUs in the system.[*] Each element is a descriptor of type cpu_workqueue_struct, whose fields are shown in Table 4-12.

[*] The reason for duplicating the work queue data structures in multiprocessor systems is that per-CPU local data structures yield a much more efficient code (see the section "Per-CPU Variables" in Chapter 5).

Table 4-12. The fields of the cpu_workqueue_struct structure

Field name	Description
lock	Spin lock used to protect the structure
remove_sequence	Sequence number used by flush_workqueue( )
insert_sequence	Sequence number used by flush_workqueue( )
worklist	Head of the list of pending functions
more_work	Wait queue where the worker thread waiting for more work to be done sleeps
work_done	Wait queue where the processes waiting for the work queue to be flushed sleep
wq	Pointer to the workqueue_struct structure containing this descriptor

4.9. Returning from Interrupts and Exceptions

We will finish the chapter by examining the termination phase of interrupt and exception handlers. (Returning from a system call is a special case, and we shall describe it in Chapter 10.) Although the main objective is clear namely, to resume execution of some program several issues must be considered before doing it:

Number of kernel control paths being concurrently executed

If there is just one, the CPU must switch back to User Mode.

Pending process switch requests

If there is any request, the kernel must perform process scheduling; otherwise, control is returned to the current process.

Pending signals

If a signal is sent to the current process, it must be handled.

Single-step mode

If a debugger is tracing the execution of the current process, single-step mode must be restored before switching back to User Mode.

Virtual-8086 mode

If the CPU is in virtual-8086 mode, the current process is executing a legacy Real Mode program, thus it must be handled in a special way.

A few flags are used to keep track of pending process switch requests, of pending signals , and of single step execution; they are stored in the flags field of the thread_info descriptor. The field stores other flags as well, but they are not related to returning from interrupts and exceptions. See Table 4-15 for a complete list of these flags.

Table 4-15. The flags field of the thread_info descriptor (continues)

Flag name	Description
TIF_SYSCALL_TRACE	System calls are being traced
TIF_NOTIFY_RESUME	Not used in the 80 x 86 platform
TIF_SIGPENDING	The process has pending signals
TIF_NEED_RESCHED	Scheduling must be performed

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.

5.1. 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.

1. If a boss calls while the waiter is idle, the waiter starts servicing the boss.
2.

2. If a boss calls while the waiter is servicing a customer, the waiter stops servicing the customer and starts servicing the boss.
3.

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.

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 80x86, 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 .

5.1.1. 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 switchfor instance, an interrupt handler that awakes a higher priority process. We will call this kind of process switch a forced process switch.

5.2. 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.

5.2.1. 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

5.3. 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

5.4. 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.

5.4.1. 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.

5.4.2. 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

Chapter 6. Timing Measurements

Countless computerized activities are driven by timing measurements , often behind the user's back. For instance, if the screen is automatically switched off after you have stopped using the computer's console, it is due to a timer that allows the kernel to keep track of how much time has elapsed since you pushed a key or moved the mouse. If you receive a warning from the system asking you to remove a set of unused files, it is the outcome of a program that identifies all user files that have not been accessed for a long time. To do these things, programs must be able to retrieve a timestamp identifying its last access time from each file. Such a timestamp must be automatically written by the kernel. More significantly, timing drives process switches along with even more visible kernel activities such as checking for time-outs.

We can distinguish two main kinds of timing measurement that must be performed by the Linux kernel:

-
- Keeping the current time and date so they can be returned to user programs through the time( ), ftime( ), and gettimeofday( ) APIs (see the section "The time( ) and gettimeofday( ) System Calls" later in this chapter) and used by the kernel itself as timestamps for files and network packets
-
- Maintaining timers mechanisms that are able to notify the kernel (see the later section "Software Timers and Delay Functions") or a user program (see the later sections "The setitimer( ) and alarm( ) System Calls" and "System Calls for POSIX Timers") that a certain interval of time has elapsed

Timing measurements are performed by several hardware circuits based on fixed-frequency oscillators and counters. This chapter consists of four different parts. The first two sections describe the hardware devices that underly timing and give an overall picture of Linux timekeeping architecture. The following sections describe the main time-related duties of the kernel: implementing CPU time sharing, updating system time and resource usage statistics, and maintaining software timers. The last section discusses the system calls related to timing measurements and the corresponding service routines.

6.1. Clock and Timer Circuits

On the 80x86 architecture, the kernel must explicitly interact with several kinds of clocks and timer circuits . The clock circuits are used both to keep track of the current time of day and to make precise time measurements. The timer circuits are programmed by the kernel, so that they issue interrupts at a fixed, predefined frequency; such periodic interrupts are crucial for implementing the software timers used by the kernel and the user programs. We'll now briefly describe the clock and hardware circuits that can be found in IBM-compatible PCs.

6.1.1. Real Time Clock (RTC)

All PCs include a clock called Real Time Clock (RTC), which is independent of the CPU and all other chips.

The RTC continues to tick even when the PC is switched off, because it is energized by a small battery. The CMOS RAM and RTC are integrated in a single chip (the Motorola 146818 or an equivalent).

The RTC is capable of issuing periodic interrupts on IRQ 8 at frequencies ranging between 2 Hz and 8,192 Hz. It can also be programmed to activate the IRQ 8 line when the RTC reaches a specific value, thus working as an alarm clock.

Linux uses the RTC only to derive the time and date; however, it allows processes to program the RTC by acting on the */dev/rtc* device file (see Chapter 13). The kernel accesses the RTC through the 0x70 and 0x71 I/O ports. The system administrator can read and write the RTC by executing the *clock* Unix system program that acts directly on these two I/O ports.

6.1.2. Time Stamp Counter (TSC)

All 80x86 microprocessors include a CLK input pin, which receives the clock signal of an external oscillator. Starting with the Pentium, 80x86 microprocessors sport a counter that is increased at each clock signal. The counter is accessible through the 64-bit Time Stamp Counter(TSC) register, which can be read by means of the rdtsc assembly language instruction. When using this register, the kernel has to take into consideration the frequency of the clock signal: if, for instance, the clock ticks at 1 GHz, the Time Stamp Counter is increased once every nanosecond.

Linux may take advantage of this register to get much more accurate time measurements than those delivered by the Programmable Interval Timer. To do this, Linux must determine the clock signal frequency while initializing the system. In fact, because this frequency is not declared when compiling the kernel, the same kernel image may run on CPUs whose clocks may tick at any frequency.

The task of figuring out the actual frequency of a CPU is accomplished during the system's boot. The calibrate_tsc( ) function computes the frequency by counting the number of clock signals that occur in a time interval of approximately 5 milliseconds. This time constant is produced by properly setting up one of the channels of the Programmable Interval Timer (see the next section).[*]

[*] To avoid losing significant digits in the integer divisions, calibrate_tsc( ) returns the duration, in microseconds, of a clock tick multiplied by 232.

6.1.3. Programmable Interval Timer (PIT)

Besides the Real Time Clock and the Time Stamp Counter, IBM-compatible PCs include another type of time-measuring device called Programmable Interval Timer(PIT). The role of a PIT is similar to the alarm clock of a microwave oven: it makes the user aware that the cooking time interval has elapsed.

6.2. The Linux Timekeeping Architecture

Linux must carry on several time-related activities. For instance, the kernel periodically:

- Updates the time elapsed since system startup.
- Updates the time and date.
- Determines, for every CPU, how long the current process has been running, and preempts it if it has exceeded the time allocated to it. The allocation of time slots (also called "quanta") is discussed in Chapter 7.
- Updates resource usage statistics.
- Checks whether the interval of time associated with each software timer (see the later section "Software Timers and Delay Functions") has elapsed.

Linux's timekeeping architecture is the set of kernel data structures and functions related to the flow of time. Actually, 80 x 86-based multiprocessor machines have a timekeeping architecture that is slightly different from the timekeeping architecture of uniprocessor machines:

- In a uniprocessor system, all time-keeping activities are triggered by interrupts raised by the global timer (either the Programmable Interval Timer or the High Precision Event Timer).
- In a multiprocessor system, all general activities (such as handling of software timers) are triggered by the interrupts raised by the global timer, while CPU-specific activities (such as monitoring the execution time of the currently running process) are triggered by the interrupts raised by the local APIC timer.

Unfortunately, the distinction between the two cases is somewhat blurred. For instance, some early SMP systems based on Intel 80486 processors didn't have local APICs. Even nowadays, there are SMP motherboards so buggy that local timer interrupts are not usable at all. In these cases, the SMP kernel must resort to the UP timekeeping architecture. On the other hand, recent uniprocessor systems feature one local APIC, so the UP kernel often makes use of the SMP timekeeping architecture. However, to simplify our description, we won't discuss these hybrid cases and will stick to the two "pure" timekeeping architectures.

Linux's timekeeping architecture depends also on the availability of the Time Stamp Counter (TSC), of the ACPI Power Management Timer, and of the High Precision Event Timer (HPET). The kernel uses two basic timekeeping functions: one to keep the current time up-to-date and another to count the number of nanoseconds that have elapsed within the current second. There are different ways to get the last value. Some methods are more precise and are available if the CPU has a Time Stamp Counter or a HPET; a less-precise method is used in the opposite case (see the later section "The time( ) and gettimeofday( ) System Calls").

6.2.1. Data Structures of the Timekeeping Architecture

6.3. Updating the Time and Date

User programs get the current time and date from the xtime variable. The kernel must periodically update this variable, so that its value is always reasonably accurate.

The update_times( ) function, which is invoked by the global timer interrupt handler, updates the value of the xtime variable as follows:

```
void update_times(void)
{
    unsigned long ticks;
    ticks = jiffies - wall_jiffies;
    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    calc_load(ticks);
}
```

We recall from the previous description of the timer interrupt handler that when the code of this function is executed, the xtime_lock seqlock has already been acquired for writing.

The wall_jiffies variable stores the time of the last update of the xtime variable. Observe that the value of wall_jiffies can be smaller than jiffies-1, since a few timer interrupts can be lost, for instance when interrupts remain disabled for a long period of time; in other words, the kernel does not necessarily update the xtime variable at every tick. However, no tick is definitively lost, and in the long run, xtime stores the correct system time. The check for lost timer interrupts is done in the mark_offset method of cur_timer; see the earlier section "Timekeeping Architecture in Uniprocessor Systems."

The update_wall_time( ) function invokes the update_wall_time_one_tick( ) function ticks consecutive times; normally, each invocation adds 1,000,000 to the xtime.tv_nsec field. If the value of xtime.tv_nsec becomes greater than 999,999,999, the update_wall_time( ) function also updates the tv_sec field of xtime. If an adjtimex( ) system call has been issued, for reasons explained in the section "The adjtimex( ) System Call" later in this chapter, the function might tune the value 1,000,000 slightly so the clock speeds up or slows down a little.

The calc_load( ) function is described in the section "Keeping Track of System Load" later in this chapter.

6.4. Updating System Statistics

The kernel, among the other time-related duties, must periodically collect various data used for:

- Checking the CPU resource limit of the running processes
- Updating statistics about the local CPU workload
- Computing the average system load
- Profiling the kernel code

6.4.1. Updating Local CPU Statistics

We have mentioned that the update_process_times( ) function is invokedeither by the global timer interrupt handler on uniprocessor systems or by the local timer interrupt handler in multiprocessor systemsto update some kernel statistics. This function performs the following steps:

1. Checks how long the current process has been running. Depending on whether the current process was running in User Mode or in Kernel Mode when the timer interrupt occurred, invokes either account_user_time( ) or account_system_time( ). Each of these functions performs essentially the following steps:
   a. Updates either the utime field (ticks spent in User Mode) or the stime field (ticks spent in Kernel Mode) of the current process descriptor. Two additional fields called cutime and cstime are provided in the process descriptor to count the number of CPU ticks spent by the process children in User Mode and Kernel Mode, respectively. For reasons of efficiency, these fields are not updated by update_process_times( ), but rather when the parent process queries the state of one of its children (see the section "Destroying Processes" in Chapter 3).
   b. Checks whether the total CPU time limit has been reached; if so, sends SIGXCPU and SIGKILL signals to current. The section "Process Resource Limits" in Chapter 3 describes how the limit is controlled by the signal->rlim[RLIMIT_CPU].rlim_cur field of each process descriptor.
   c. Invokes account_it_virt( ) and account_it_prof( ) to check the process timers (see the section "The setitimer( ) and alarm( ) System Calls" later in this chapter).
   d. Updates some kernel statistics stored in the kstat per-CPU variable.
2. Invokes raise_softirq( ) to activate the TIMER_SOFTIRQ tasklet on the local CPU (see the section "Software Timers and Delay Functions" later in this chapter).
3.

6.5. Software Timers and Delay Functions

A timer is a software facility that allows functions to be invoked at some future moment, after a given time interval has elapsed; a time-out denotes a moment at which the time interval associated with a timer has elapsed.

Timers are widely used both by the kernel and by processes. Most device drivers use timers to detect anomalous conditions floppy disk drivers, for instance, use timers to switch off the device motor after the floppy has not been accessed for a while, and parallel printer drivers use them to detect erroneous printer conditions.

Timers are also used quite often by programmers to force the execution of specific functions at some future time (see the later section "The setitimer( ) and alarm( ) System Calls").

Implementing a timer is relatively easy. Each timer contains a field that indicates how far in the future the timer should expire. This field is initially calculated by adding the right number of ticks to the current value of jiffies. The field does not change. Every time the kernel checks a timer, it compares the expiration field to the value of jiffies at the current moment, and the timer expires when jiffies is greater than or equal to the stored value.

Linux considers two types of timers called dynamic timers and interval timers . The first type is used by the kernel, while interval timers may be created by processes in User Mode.

One word of caution about Linux timers: since checking for timer functions is always done by deferrable functions that may be executed a long time after they have been activated, the kernel cannot ensure that timer functions will start right at their expiration times. It can only ensure that they are executed either at the proper time or after with a delay of up to a few hundreds of milliseconds. For this reason, timers are not appropriate for real-time applications in which expiration times must be strictly enforced.

Besides software timers , the kernel also makes use of delay functions , which execute a tight instruction loop until a given time interval elapses. We will discuss them in the later section "Delay Functions."

6.5.1. Dynamic Timers

Dynamic timers may be dynamically created and destroyed. No limit is placed on the number of currently active dynamic timers.

A dynamic timer is stored in the following timer_list structure:

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    spinlock_t lock;
    unsigned long magic;
    void (*function)(unsigned long);
    unsigned long data;
    tvec_base_t *base;
};
```

The function field contains the address of the function to be executed when the timer expires. The data field specifies a parameter to be passed to this timer function. Thanks to the data field, it is possible to define a single general-purpose function that handles the time-outs of several device drivers; the data field could store the device ID or other meaningful data that could be used by the function to differentiate the device.

6.6. System Calls Related to Timing Measurements

Several system calls allow User Mode processes to read and modify the time and date and to create timers. Let's briefly review these and discuss how the kernel handles them.

6.6.1. The time( ) and gettimeofday( ) System Calls

Processes in User Mode can get the current time and date by means of several system calls:

time( )

Returns the number of elapsed seconds since midnight at the start of January 1, 1970 (UTC).

gettimeofday( )

Returns, in a data structure named timeval, the number of elapsed seconds since midnight of January 1, 1970 (UTC) and the number of elapsed microseconds in the last second (a second data structure named timezone is not currently used).

The time( ) system call is superseded by gettimeofday( ), but it is still included in Linux for backward compatibility. Another widely used function, ftime( ), which is no longer implemented as a system call, returns the number of elapsed seconds since midnight of January 1, 1970 (UTC) and the number of elapsed milliseconds in the last second.

The gettimeofday( ) system call is implemented by the sys_gettimeofday( ) function. To compute the current date and time of the day, this function invokes do_gettimeofday( ), which executes the following actions:

1. Acquires the xtime_lock seqlock for reading.
2. Determines the number of microseconds elapsed since the last timer interrupt by invoking the get_offset method of the cur_timer timer object:

```
usec = cur_timer->getoffset( );
```

3. As explained in the earlier section "Data Structures of the Timekeeping Architecture," there are four possible cases:
   a. If cur_timer points to the timer_hpet object, the method compares the current value of the HPET counter with the value of the same counter saved in the last execution of the timer interrupt handler.
   b. If cur_timer points to the timer_pmtmr object, the method compares the current value of the ACPI PMT counter with the value of the same counter saved in the last execution of the timer interrupt handler.
   c.

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 x 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.

7.1. Scheduling Policy

The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and so on. The set of rules used to determine when and how 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 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.

[*] Recall that stopped and suspended processes cannot be selected by the scheduling algorithm to run on a CPU.

The scheduling policy is also based on ranking processes according to their priority. Complicated algorithms are sometimes used to derive the current priority of a process, but the end result is the same: each process is associated with a value that 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.

7.2. 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 costlythat is, it spends too much timein 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.

7.2.1. 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( )

7.3. 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 processesthat is, of those in a TASK_RUNNING stateexcept the swapper process (idle process).

7.3.1. 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

7.4. 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

7.4.1. 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.

7.5. 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

7.6. 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.

7.6.1. 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.

[*] 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.

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.

7.6.2. The getpriority( ) and setpriority( ) System Calls

The nice( ) system call affects only the process that invokes it. Two other system calls, denoted as getpriority( ) and setpriority( ), act on the base priorities of all processes in a given group. getpriority( ) returns 20 minus the lowest nice field value among all processes in a given groupthat is, the highest priority among those 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

Chapter 8. Memory Management

We saw in Chapter 2 how Linux takes advantage of 80 x 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.

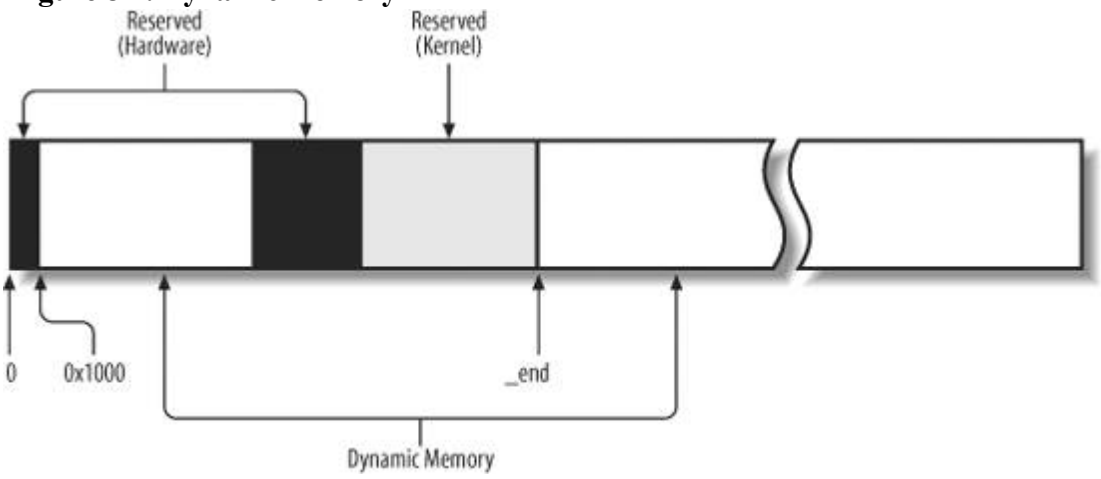
This chapter, which consists of three main sections, describes how the kernel allocates dynamic memory for its own use. The sections "Page Frame Management" and "Memory Area Management" illustrate two different techniques for handling physically contiguous memory areas, while the section "Noncontiguous 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.

8.1. 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 enabledsee 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



8.1.1. 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

8.2. Memory Area Management

This section deals with memory areas that is, with sequences of memory cells having contiguous physical addresses and an arbitrary length.

The buddy system algorithm adopts the page frame as the basic memory area. This is fine for dealing with relatively large memory requests, but how are we going to deal with requests for small memory areas, say a few tens or 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.

8.2.1. 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.
-

8.3. Noncontiguous Memory Area Management

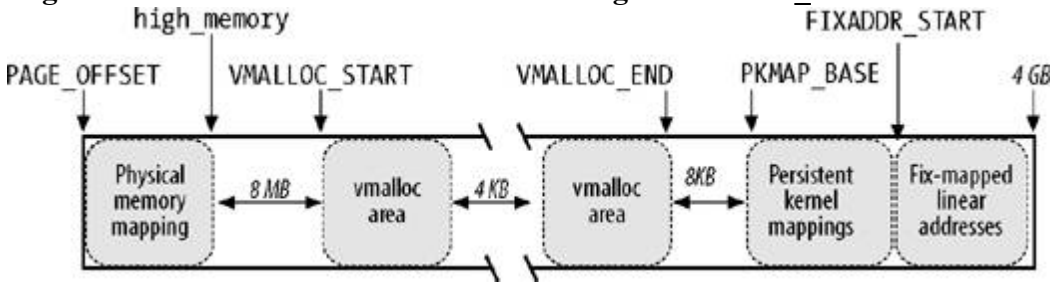
We already know that it is preferable to map memory areas into sets of contiguous page frames, thus making better use of the cache and achieving lower average memory access times. Nevertheless, if the requests for memory areas are infrequent, it makes sense to consider an allocation scheme based on noncontiguous 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").

8.3.1. Linear Addresses of Noncontiguous Memory Areas

To find a free range of linear addresses, we can look in the area starting from PAGE_OFFSET (usually 0xc0000000, the beginning of the fourth gigabyte). 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

Chapter 9. Process Address Space

As seen in the previous chapter, a kernel function gets dynamic memory in a fairly straightforward manner by invoking one of a variety of functions: _ _get_free_pages( ) or alloc_pages( ) to get pages from the zoned page frame allocator, kmem_cache_alloc( ) or kmalloc( ) to use the slab allocator for specialized or general-purpose objects, and vmalloc( ) or vmalloc_32( ) to get a noncontiguous memory area. If the request can be satisfied, each of these functions returns a page descriptor address or a linear address identifying the beginning of the allocated dynamic memory area.

These simple approaches work for two reasons:

- The kernel is the highest-priority component of the operating system. If a kernel function makes a request for dynamic memory, it must have a valid reason to issue that request, and there is no point in trying to defer it.
- The kernel trusts itself. All kernel functions are assumed to be error-free, so the kernel does not need to insert any protection against programming errors.

When allocating memory to User Mode processes, the situation is entirely different:

- Process requests for dynamic memory are considered non-urgent. When a process's executable file is loaded, for instance, it is unlikely that the process will address all the pages of code in the near future. Similarly, when a process invokes malloc( ) to get additional dynamic memory, it doesn't mean the process will soon access all the additional memory obtained. Thus, as a general rule, the kernel tries to defer allocating dynamic memory to User Mode processes.
- Because user programs cannot be trusted, the kernel must be prepared to catch all addressing errors caused by processes in User Mode.

As this chapter describes, the kernel succeeds in deferring the allocation of dynamic memory to processes by using a new kind of resource. When a User Mode process asks for dynamic memory, it doesn't get additional page frames; instead, it gets the right to use a new range of linear addresses, which become part of its address space. This interval is called a "memory region."

In the next section, we discuss how the process views dynamic memory. We then describe the basic components of the process address space in the section "Memory Regions." Next, we examine in detail the role played by the Page Fault exception handler in deferring the allocation of page frames to processes and illustrate how the kernel creates and deletes whole process address spaces. Last, we discuss the APIs and system calls related to address space management.

9.1. The Process's Address Space

The address space of a process consists of all linear addresses that the process is allowed to use. Each process sees a different set of linear addresses; the address used by one process bears no relation to the address used by another. As we will see later, the kernel may dynamically modify a process address space by adding or removing intervals of linear addresses.

The kernel represents intervals of linear addresses by means of resources called memory regions , which are characterized by an initial linear address, a length, and some access rights. For reasons of efficiency, both the initial address and the length of a memory region must be multiples of 4,096, so that the data identified by each memory region completely fills up the page frames allocated to it. Following are some typical situations in which a process gets new memory regions:

- When the user types a command at the console, the shell process creates a new process to execute the command. As a result, a fresh address space, and thus a set of memory regions, is assigned to the new process (see the section "Creating and Deleting a Process Address Space" later in this chapter; also, see Chapter 20).
- A running process may decide to load an entirely different program. In this case, the process ID remains unchanged, but the memory regions used before loading the program are released and a new set of memory regions is assigned to the process (see the section "The exec Functions" in Chapter 20).
- A running process may perform a "memory mapping" on a file (or on a portion of it). In such cases, the kernel assigns a new memory region to the process to map the file (see the section " Memory Mapping" in Chapter 16).
- A process may keep adding data on its User Mode stack until all addresses in the memory region that map the stack have been used. In this case, the kernel may decide to expand the size of that memory region (see the section "Page Fault Exception Handler" later in this chapter).
- A process may create an IPC-shared memory region to share data with other cooperating processes. In this case, the kernel assigns a new memory region to the process to implement this construct (see the section "IPC Shared Memory" in Chapter 19).
- A process may expand its dynamic area (the heap) through a function such as malloc( ). As a result, the kernel may decide to expand the size of the memory region assigned to the heap (see the section "Managing the Heap" later in this chapter).

Table 9-1 illustrates some of the system calls related to the previously mentioned tasks. brk( ) is discussed at the end of this chapter, while the remaining system calls are described in other chapters.

Table 9-1. System calls related to memory region creation and deletion

System call	Description
brk( )	Changes the heap size of the process

9.2. The Memory Descriptor

All information related to the process address space is included in an object called the memory descriptor of type mm_struct. This object is referenced by the mm field of the process descriptor. The fields of a memory descriptor are listed in Table 9-2.

Table 9-2. The fields of the memory descriptor

Type	Field	Description
struct vm_area_struct *	mmap	Pointer to the head of the list of memory region objects
struct rb_root	mm_rb	Pointer to the root of the red-black tree of memory region objects
struct vm_area_struct *	mmap_cache	Pointer to the last referenced memory region object
unsigned long (*)( )	get_unmapped_area	Method that searches an available linear address interval in the process address space
void (*)( )	unmap_area	Method invoked when releasing a linear address interval
unsigned long	mmap_base	Identifies the linear address of the first allocated anonymous memory region or file memory mapping (see the section "Program Segments and Process Memory Regions" in Chapter 20)
unsigned long	free_area_cache	Address from which the kernel will look for a free interval of linear addresses in the process address space
pgd_t *	pgd	Pointer to the Page Global Directory
atomic_t	mm_users	Secondary usage counter
atomic_t	mm_count	Main usage counter
int	map_count	Number of memory regions

9.3. Memory Regions

Linux implements a memory region by means of an object of type vm_area_struct; its fields are shown in Table 9-3.[*]

[*] We omitted describing a few additional fields used in NUMA systems.

Table 9-3. The fields of the memory region object

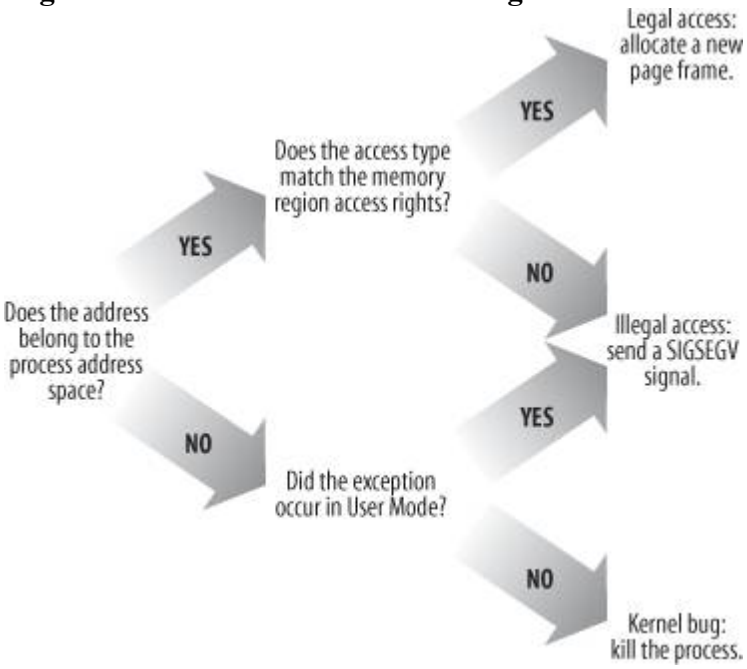
Type	Field	Description
struct mm_struct *	vm_mm	Pointer to the memory descriptor that owns the region.
unsigned long	vm_start	First linear address inside the region.
unsigned long	vm_end	First linear address after the region.
struct vm_area_struct *	vm_next	Next region in the process list.
pgprot_t	vm_page_prot	Access permissions for the page frames of the region.
unsigned long	vm_flags	Flags of the region.
struct rb_node	vm_rb	Data for the red-black tree (see later in this chapter).
union	shared	Links to the data structures used for reverse mapping (see the section "Reverse Mapping for Mapped Pages" in Chapter 17).
struct list_head	anon_vma_node	Pointers for the list of anonymous memory regions (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
struct anon_vma *	anon_vma	Pointer to the anon_vma data structure (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).

9.4. Page Fault Exception Handler

As stated previously, the Linux Page Fault exception handler must distinguish exceptions caused by programming errors from those caused by a reference to a page that legitimately belongs to the process address space but simply hasn't been allocated yet.

The memory region descriptors allow the exception handler to perform its job quite efficiently. The do_page_fault( ) function, which is the Page Fault interrupt service routine for the 80 x 86 architecture, compares the linear address that caused the Page Fault against the memory regions of the current process; it can thus determine the proper way to handle the exception according to the scheme that is illustrated in Figure 9-4.

Figure 9-4. Overall scheme for the Page Fault handler



In practice, things are a lot more complex because the Page Fault handler must recognize several particular subcases that fit awkwardly into the overall scheme, and it must distinguish several kinds of legal access. A detailed flow diagram of the handler is illustrated in Figure 9-5.

The identifiers vmalloc_fault, good_area, bad_area, and no_context are labels appearing in do_page_fault( ) that should help you to relate the blocks of the flow diagram to specific lines of code.

The do_ page_fault( ) function accepts the following input parameters:

-
- The regs address of a pt_regs structure containing the values of the microprocessor registers when the exception occurred.
-
- A 3-bit error_code, which is pushed on the stack by the control unit when the exception occurred (see "Hardware Handling of Interrupts and Exceptions" in Chapter 4). The bits have the following meanings:
  -
  - If bit 0 is clear, the exception was caused by an access to a page that is not present (the

9.5. Creating and Deleting a Process Address Space

Of the six typical cases mentioned earlier in the section "The Process's Address Space," in which a process gets new memory regions, the first oneissuing a fork( ) system callrequires the creation of a whole new address space for the child process. Conversely, when a process terminates, the kernel destroys its address space. In this section, we discuss how these two activities are performed by Linux.

9.5.1. Creating a Process Address Space

In the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3, we mentioned that the kernel invokes the copy_mm( ) function while creating a new process. This function creates the process address space by setting up all Page Tables and memory descriptors of the new process.

Each process usually has its own address space, but lightweight processes can be created by calling clone( ) with the CLONE_VM flag set. These processes share the same address space; that is, they are allowed to address the same set of pages.

Following the COW approach described earlier, traditional processes inherit the address space of their parent: pages stay shared as long as they are only read. When one of the processes attempts to write one of them, however, the page is duplicated; after some time, a forked process usually gets its own address space that is different from that of the parent process. Lightweight processes, on the other hand, use the address space of their parent process. Linux implements them simply by not duplicating address space. Lightweight processes can be created considerably faster than normal processes, and the sharing of pages can also be considered a benefit as long as the parent and children coordinate their accesses carefully.

If the new process has been created by means of the clone( ) system call and if the CLONE_VM flag of the flag parameter is set, copy_mm( ) gives the clone (tsk) the address space of its parent (current):

```
if (clone_flags & CLONE_VM) {
    atomic_inc(&current->mm->mm_users);
    spin_unlock_wait(&current->mm->page_table_lock);
    tsk->mm = current->mm;
    tsk->active_mm = current->mm;
    return 0;
}
```

Invoking the spin_unlock_wait( ) function ensures that, if the page table spin lock of the process is held by some other CPU, the page fault handler does not terminate until that lock is released. In fact, beside protecting the page tables, this spin lock must forbid the creation of new lightweight processes sharing the current->mm descriptor.

If the CLONE_VM flag is not set, copy_mm( ) must create a new address space (even though no memory is allocated within that address space until the process requests an address). The function allocates a new memory descriptor, stores its address in the mm field of the new process descriptor tsk, and copies the contents of current->mm into tsk->mm. It then changes a few fields of the new descriptor:

```
tsk->mm = kmem_cache_alloc(mm_cachep, SLAB_KERNEL);
memcpy(tsk->mm, current->mm, sizeof(*tsk->mm));
atomic_set(&tsk->mm->mm_users, 1);
atomic_set(&tsk->mm->mm_count, 1);
init_rwsem(&tsk->mm->mmap_sem);
tsk->mm->core_waiters = 0;
tsk->mm->page_table_lock = SPIN_LOCK_UNLOCKED;
tsk->mm->ioctx_list_lock = RW_LOCK_UNLOCKED;
tsk->mm->ioctx_list = NULL;
```

9.6. Managing the Heap

Each Unix process owns a specific memory region called the heap, which is used to satisfy the process's dynamic memory requests. The start_brk and brk fields of the memory descriptor delimit the starting and ending addresses, respectively, of that region.

The following APIs can be used by the process to request and release dynamic memory:

malloc(size)

Requests size bytes of dynamic memory; if the allocation succeeds, it returns the linear address of the first memory location.

calloc(n,size)

Requests an array consisting of n elements of size size; if the allocation succeeds, it initializes the array components to 0 and returns the linear address of the first element.

realloc(ptr,size)

Changes the size of a memory area previously allocated by malloc( ) or calloc( ) .

free(addr)

Releases the memory region allocated by malloc( ) or calloc( ) that has an initial address of addr.

brk(addr)

Modifies the size of the heap directly; the addr parameter specifies the new value of current->mm->brk, and the return value is the new ending address of the memory region (the process must check whether it coincides with the requested addr value).

sbrk(incr)

Is similar to brk( ) , except that the incr parameter specifies the increment or decrement of the heap size in bytes.

The brk( ) function differs from the other functions listed because it is the only one implemented as a system call. All the other functions are implemented in the C library by using brk( ) and mmap( ).[*]

[*] The realloc( ) library function can also make use of the mremap( ) system call.

When a process in User Mode invokes the brk( ) system call, the kernel executes the sys_brk(addr) function. This function first verifies whether the addr parameter falls inside the memory region that contains the process code; if so, it returns immediately because the heap cannot overlap with memory region containing the process's code:

```
mm = current->mm;
```

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.

10.1. 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 wayan 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 irrelevantthe 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 x 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.

10.2. System Call Handler and Service Routines

When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function. As we will see in the next section, in the 80 x 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.

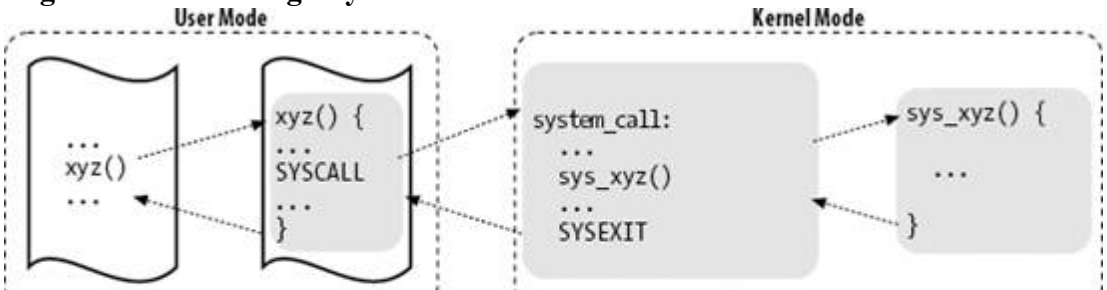
The system call handler, which has a structure similar to that of the other exception handlers, performs the following operations:

-
- 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



10.3. Entering and Exiting a System Call

Native applications[*] can invoke a system call in two different ways:

[*] 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).

- 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 callthus switching the CPU back to User Modein 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.

10.3.1. 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 128in hexadecimal, 0x80is 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:

10.4. 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).[*]
- [*] We refer, as usual, to the 32-bit architecture of the 80 x 86 processors. The discussion in this section does not apply to 64-bit architectures.
-
- The number of parameters must not exceed six, besides the system call number passed in eax, because 80 x 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

10.5. 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 x 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
```

Chapter 11. Signals

Signals were introduced by the first Unix systems to allow interactions between User Mode processes; the kernel also uses them to notify processes of system events. Signals have been around for 30 years with only minor changes.

The first sections of this chapter examine in detail how signals are handled by the Linux kernel, then we discuss the system calls that allow processes to exchange signals.

PREV NEXT

11.1. The Role of Signals

A signal is a very short message that may be sent to a process or a group of processes. The only information given to the process is usually a number identifying the signal; there is no room in standard signals for arguments, a message, or other accompanying information.

A set of macros whose names start with the prefix SIG is used to identify signals; we have already made a few references to them in previous chapters. For instance, the SIGCHLD macro was mentioned in the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3. This macro, which expands into the value 17 in Linux, yields the identifier of the signal that is sent to a parent process when a child stops or terminates. The SIGSEGV macro, which expands into the value 11, was mentioned in the section "Page Fault Exception Handler" in Chapter 9; it yields the identifier of the signal that is sent to a process when it makes an invalid memory reference.

Signals serve two main purposes:

-
- To make a process aware that a specific event has occurred
-
- To cause a process to execute a signal handler function included in its code

Of course, the two purposes are not mutually exclusive, because often a process must react to some event by executing a specific routine.

Table 11-1 lists the first 31 signals handled by Linux 2.6 for the 80x86 architecture (some signal numbers, such those associated with SIGCHLD or SIGSTOP, are architecture-dependent; furthermore, some signals such as SIGSTKFLT are defined only for specific architectures). The meanings of the default actions are described in the next section.

Table 11-1. The first 31 signals in Linux/i386

#	Signal name	Default action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard	Yes
3	SIGQUIT	Dump	Quit from keyboard	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes

11.2. Generating a Signal

Many kernel functions generate signals: they accomplish the first phase of signal handlingdescribed earlier in the section "The Role of Signals"by updating one or more process descriptors as needed. They do not directly perform the second phase of delivering the signal but, depending on the type of signal and the state of the destination processes, may wake up some processes and force them to receive the signal.

When a signal is sent to a process, either from the kernel or from another process, the kernel generates it by invoking one of the functions listed in Table 11-9.

Table 11-9. Kernel functions that generate a signal for a process

Name	Description
send_sig( )	Sends a signal to a single process
send_sig_info( )	Like send_sig( ), with extended information in a siginfo_t structure
force_sig( )	Sends a signal that cannot be explicitly ignored or blocked by the process
force_sig_info( )	Like force_sig( ), with extended information in a siginfo_t structure
force_sig_specific( )	Like force_sig( ), but optimized for SIGSTOP and SIGKILL signals
sys_tkill( )	System call handler of tkill( ) (see the later section "System Calls Related to Signal Handling")
sys_tgkill( )	System call handler of tgkill( )

All functions in Table 11-9 end up invoking the specific_send_sig_info( ) function described in the next section.

When a signal is sent to a whole thread group, either from the kernel or from another process, the kernel generates it by invoking one of the functions listed in Table 11-10.

Table 11-10. Kernel functions that generate a signal for a thread group

Name	Description
send_group_sig_info( )	Sends a signal to a single thread group identified by the process descriptor of one of its members
kill_pg( )	Sends a signal to all thread groups in a process group (see the section "Process Management" in

11.3. Delivering a Signal

We assume that the kernel noticed the arrival of a signal and invoked one of the functions mentioned in the previous sections to prepare the process descriptor of the process that is supposed to receive the signal. But in case that process was not running on the CPU at that moment, the kernel deferred the task of delivering the signal. We now turn to the activities that the kernel performs to ensure that pending signals of a process are handled.

As mentioned in the section "Returning from Interrupts and Exceptions" in Chapter 4, the kernel checks the value of the TIF_SIGPENDING flag of the process before allowing the process to resume its execution in User Mode. Thus, the kernel checks for the existence of pending signals every time it finishes handling an interrupt or an exception.

To handle the nonblocked pending signals, the kernel invokes the do_signal( ) function, which receives two parameters:

regs

The address of the stack area where the User Mode register contents of the current process are saved.

oldset

The address of a variable where the function is supposed to save the bit mask array of blocked signals. It is NULL if there is no need to save the bit mask array.

Our description of the do_signal( ) function will focus on the general mechanism of signal delivery; the actual code is burdened with lots of details dealing with race conditions and other special casessuch as freezing the system, generating core dumps, stopping and killing a whole thread group, and so on. We will quietly skip all these details.

As already mentioned, the do_signal( ) function is usually only invoked when the CPU is going to return in User Mode. For that reason, if an interrupt handler invokes do_signal( ), the function simply returns:

```
if ((regs->xcs & 3) != 3)
    return 1;
```

If the oldset parameter is NULL, the function initializes it with the address of the current->blocked field:

```
if (!oldset)
    oldset = &current->blocked;
```

The heart of the do_signal( ) function consists of a loop that repeatedly invokes the dequeue_signal( ) function until no nonblocked pending signals are left in both the private and shared pending signal queues. The return code of dequeue_signal( ) is stored in the signr local variable. If its value is 0, it means that all pending signals have been handled and do_signal( ) can finish. As long as a nonzero value is returned, a pending signal is waiting to be handled. dequeue_signal( ) is invoked again after do_signal( ) handles the current signal.

The dequeue_signal( ) considers first all signals in the private pending signal queue, starting from the lowest-numbered signal, then the signals in the shared queue. It updates the data structures to indicate that the signal is no longer pending and returns its number. This task involves clearing the corresponding bit in current->pending.signal or current->signal->shared_pending.signal, and invoking recalc_sigpending( ) to update the value of the TIF_SIGPENDING flag.

11.4. System Calls Related to Signal Handling

As stated in the introduction of this chapter, programs running in User Mode are allowed to send and receive signals. This means that a set of system calls must be defined to allow these kinds of operations. Unfortunately, for historical reasons, several system calls exist that serve essentially the same purpose. As a result, some of these system calls are never invoked. For instance, sys_sigaction( ) and sys_rt_sigaction( ) are almost identical, so the sigaction( ) wrapper function included in the C library ends up invoking sys_rt_sigaction( ) instead of sys_sigaction( ). We will describe some of the most significant system calls in the following sections.

11.4.1. The kill( ) System Call

The kill(pid,sig) system call is commonly used to send signals to conventional processes or multithreaded applications; its corresponding service routine is the sys_kill( ) function. The integer pid parameter has several meanings, depending on its numerical value:

pid > 0

The sig signal is sent to the thread group of the process whose PID is equal to pid.

pid = 0

The sig signal is sent to all thread groups of the processes in the same process group as the calling process.

pid = -1

The signal is sent to all processes, except swapper (PID 0), init (PID 1), and current.

pid < -1

The signal is sent to all thread groups of the processes in the process group -pid.

The sys_kill( ) function sets up a minimal siginfo_t table for the signal, and then invokes kill_something_info( ):

```
info.si_signo = sig;
info.si_errno = 0;
info.si_code = SI_USER;
info._sifields._kill._pid = current->tgid;
info._sifields._kill._uid = current->uid;
return kill_something_info(sig, &info, pid);
```

The kill_something_info( ) function, in turn, invokes either kill_proc_info( ) (to send the signal to a single thread group via group_send_sig_info( )), or kill_pg_info( ) (to scan all processes in the destination process group and invoke send_sig_info( ) for each of them), or repeatedly group_send_sig_info( ) for each process in the system (if pid is -1).

The kill( ) system call is able to send every signal, even the so-called real-time signals that have numbers ranging from 32 to 64. However, as we saw in the earlier section "Generating a Signal," the kill( ) system call does not ensure that a new element is added to the pending signal queue of the destination process

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 typesnamely, 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.

12.1. The Role of the Virtual Filesystem (VFS)

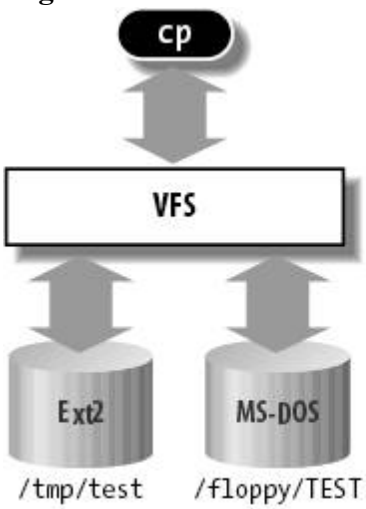
The Virtual Filesystem (also known as Virtual Filesystem Switch or VFS) is a kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.

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).

Figure 12-1. VFS role in a simple file copy operation




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

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)[*]
- [*] Although these filesystems owe their birth to Linux, they have been ported to several other operating systems.
-
- Filesystems for Unix variants such as sysv filesystem (System V , Coherent , Xenix), UFS (BSD , Solaris , NEXTSTEP), MINIX filesystem, and VERITAS VxFS (SCO UnixWare)

12.2. VFS Data Structures

Each VFS object is stored in a suitable data structure, which includes both the object attributes and a pointer to a table of object methods. The kernel may dynamically modify the methods of the object and, hence, it may install specialized behavior for the object. The following sections explain the VFS objects and their interrelationships in detail.

12.2.1. Superblock Objects

A superblock object consists of a super_block structure whose fields are described in Table 12-2.

Table 12-2. The fields of the superblock object

Type	Field	Description
struct list_head	s_list	Pointers for superblock list
dev_t	s_dev	Device identifier
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

12.3. 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 registrationthat 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.

12.3.1. 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

12.4. 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 mountedeither by the initialization scripts or directly by the userson 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.[*]

[*] 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.

12.4.1. 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 filesystemsthe 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 mountsor unmountsa 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

12.5. 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.

12.6. 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.

12.6.1. 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 descriptorthat 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

12.7. 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.[*]

[*] Oddly enough, a process may still unlink (delete) a file even if some other process owns a mandatory lock on it! This perplexing situation is possible because when a process deletes a file hard link, it does not modify its contents, but only the contents of its parent directory.

Whether processes use advisory or mandatory locks, they can 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.

12.7.1. 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

Chapter 13. I/O Architecture and Device Drivers

The Virtual File System in the last chapter depends on lower-level functions to carry out each read, write, or other operation in a manner suited to each device. The previous chapter included a brief discussion of how operations are handled by different filesystems. In this chapter, we look at how the kernel invokes the operations on actual devices.

In the section "I/O Architecture," we give a brief survey of the 80 x 86 I/O architecture. In the section "The Device Driver Model," we introduce the Linux device driver model. Next, in the section "Device Files," we show how the VFS associates a special file called "device file" with each different hardware device, so that application programs can use all kinds of devices in the same way. We then introduce in the section "Device Drivers" some common characteristics of device drivers. Finally, in the section "Character Device Drivers," we illustrate the overall organization of character device drivers in Linux. We'll defer the discussion of block device drivers to the next chapters.

Readers interested in developing device drivers on their own may want to refer to Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman's Linux Device Drivers, Third Edition (O'Reilly).

13.1. I/O Architecture

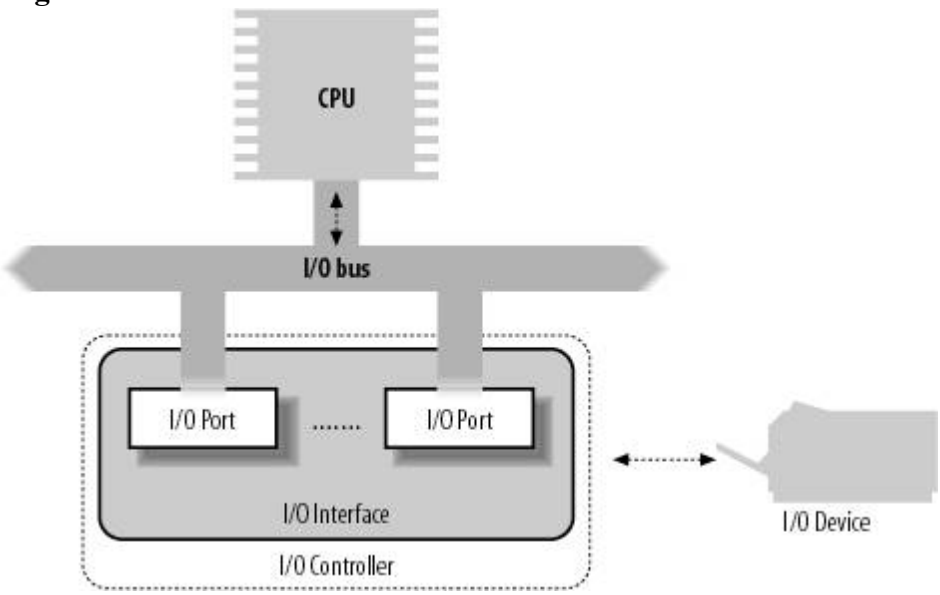
To make a computer work properly, data paths must be provided that let information flow between CPU(s), RAM, and the score of I/O devices that can be connected to a personal computer. These data paths, which are denoted as the buses , act as the primary communication channels inside the computer.

Any computer has a system bus that connects most of the internal hardware devices. A typical system bus is the PCI (Peripheral Component Interconnect) bus. Several other types of buses, such as ISA, EISA, MCA, SCSI, and USB, are currently in use. Typically, the same computer includes several buses of different types, linked together by hardware devices called bridges . Two high-speed buses are dedicated to the data transfers to and from the memory chips: the frontside bus connects the CPUs to the RAM controller, while the backside bus connects the CPUs directly to the external hardware cache. The host bridge links together the system bus and the frontside bus.

Any I/O device is hosted by one, and only one, bus. The bus type affects the internal design of the I/O device, as well as how the device has to be handled by the kernel. In this section, we discuss the functional characteristics common to all PC architectures, without giving details about a specific bus type.

The data path that connects a CPU to an I/O device is generically called an I/O bus. The 80 x 86 microprocessors use 16 of their address pins to address I/O devices and 8, 16, or 32 of their data pins to transfer data. The I/O bus, in turn, is connected to each I/O device by means of a hierarchy of hardware components including up to three elements: I/O ports , interfaces, and device controllers. Figure 13-1 shows the components of the I/O architecture.

Figure 13-1. PC's I/O architecture



13.1.1. I/O Ports

Each device connected to the I/O bus has its own set of I/O addresses, which are usually called I/O ports. In the IBM PC architecture, the I/O address space provides up to 65,536 8-bit I/O ports. Two consecutive 8-bit ports may be regarded as a single 16-bit port, which must start on an even address. Similarly, two consecutive 16-bit ports may be regarded as a single 32-bit port, which must start on an address that is a multiple of 4. Four special assembly language instructions called in, ins , out , and outs allow the CPU to read from and write into an I/O port. While executing one of these instructions, the CPU selects the required I/O port and transfers the data between a CPU register and the port.

13.2. The Device Driver Model

Earlier versions of the Linux kernel offered few basic functionalities to the device driver developers: allocating dynamic memory, reserving a range of I/O addresses or an IRQ line, activating an interrupt service routine in response to a device's interrupt. Older hardware devices, in fact, were cumbersome and difficult to program, and two different hardware devices had little in common even if they were hosted on the same bus. Thus, there was no point in trying to offer a unifying model to the device driver developers.

Things are different now. Bus types such as PCI put strong demands on the internal design of the hardware devices; as a consequence, recent hardware devices, even of different classes, sport similar functionalities. Drivers for such devices should typically take care of:

-
- Power management (handling of different voltage levels on the device's power line)
-
- Plug and play (transparent allocation of resources when configuring the device)
-
- Hot-plugging (support for insertion and removal of the device while the system is running)

Power management is performed globally by the kernel on every hardware device in the system. For instance, when a battery-powered computer enters the "standby" state, the kernel must force every hardware device (hard disks, graphics card, sound card, network card, bus controllers, and so on) in a low-power state. Thus, each driver of a device that can be put in the "standby" state must include a callback function that puts the hardware device in the low-power state. Moreover, the hardware devices must be put in the "standby" state in a precise order, otherwise some devices could be left in the wrong power state. For instance, the kernel must put in "standby" first the hard disks and then their disk controller, because in the opposite case it would be impossible to send commands to the hard disks.

To implement these kinds of operations, Linux 2.6 provides some data structures and helper functions that offer a unifying view of all buses, devices, and device drivers in the system; this framework is called the device driver model .

13.2.1. The sysfs Filesystem

The sysfs filesystem is a special filesystem similar to */proc* that is usually mounted on the */sys* directory. The */proc* filesystem was the first special filesystem designed to allow User Mode applications to access kernel internal data structures. The */sysfs* filesystem has essentially the same objective, but it provides additional information on kernel data structures; furthermore, */sysfs* is organized in a more structured way than */proc*. Likely, both */proc* and */sysfs* will continue to coexist in the near future.

A goal of the sysfs filesystem is to expose the hierarchical relationships among the components of the device driver model. The related top-level directories of this filesystem are:

block

The block devices, independently from the bus to which they are connected.

devices

13.3. Device Files

As mentioned in Chapter 1, Unix-like operating systems are based on the notion of a file, which is just an information container structured as a sequence of bytes. According to this approach, I/O devices are treated as special files called device files ; thus, the same system calls used to interact with regular files on disk can be used to directly interact with I/O devices. For example, the same write( ) system call may be used to write data into a regular file or to send it to a printer by writing to the */dev/lp0* device file.

According to the characteristics of the underlying device drivers, device files can be of two types: block or character. The difference between the two classes of hardware devices is not so clear-cut. At least we can assume the following:

-
- The data of a block device can be addressed randomly, and the time needed to transfer a data block is small and roughly the same, at least from the point of view of the human user. Typical examples of block devices are hard disks, floppy disks , CD-ROM drives, and DVD players.
-
- The data of a character device either cannot be addressed randomly (consider, for instance, a sound card), or they can be addressed randomly, but the time required to access a random datum largely depends on its position inside the device (consider, for instance, a magnetic tape driver).

Network cards are a notable exception to this schema, because they are hardware devices that are not directly associated with device files.

Device files have been in use since the early versions of the Unix operating system. A device file is usually a real file stored in a filesystem. Its inode, however, doesn't need to include pointers to blocks of data on the disk (the file's data) because there are none. Instead, the inode must include an identifier of the hardware device corresponding to the character or block device file.

Traditionally, this identifier consists of the type of device file (character or block) and a pair of numbers. The first number, called the major number, identifies the device type. Traditionally, all device files that have the same major number and the same type share the same set of file operations, because they are handled by the same device driver. The second number, called the minor number, identifies a specific device among a group of devices that share the same major number. For instance, a group of disks managed by the same disk controller have the same major number and different minor numbers .

The mknod( ) system call is used to create device files. It receives the name of the device file, its type, and the major and minor numbers as its parameters. Device files are usually included in the */dev* directory. Table 13-7 illustrates the attributes of some device files. Notice that character and block devices have independent numbering, so block device (3,0) is different from character device (3,0).

Table 13-7. Examples of device files

Name	Type	Major	Minor	Description
/dev/fd0	block	2	0	Floppy disk
/dev/hda	block	3	0	First IDE disk
				Second primary

13.4. Device Drivers

A device driver is the set of kernel routines that makes a hardware device respond to the programming interface defined by the canonical set of VFS functions (*open*, *read*, *lseek*, *ioctl*, and so forth) that control a device. The actual implementation of all these functions is delegated to the device driver. Because each device has a different I/O controller, and thus different commands and different state information, most I/O devices have their own drivers.

There are many types of device drivers . They mainly differ in the level of support that they offer to the User Mode applications, as well as in their buffering strategies for the data collected from the hardware devices. Because these choices greatly influence the internal structure of a device driver, we discuss them in the sections "Direct Memory Access (DMA)" and "Buffering Strategies for Character Devices."

A device driver does not consist only of the functions that implement the device file operations. Before using a device driver, several activities must have taken place. We'll examine them in the following sections.

13.4.1. Device Driver Registration

We know that each system call issued on a device file is translated by the kernel into an invocation of a suitable function of a corresponding device driver. To achieve this, a device driver must register itself. In other words, registering a device driver means allocating a new device_driver descriptor, inserting it in the data structures of the device driver model (see the earlier section "Components of the Device Driver Model"), and linking it to the corresponding device file(s). Accesses to device files whose corresponding drivers have not been previously registered return the error code -ENODEV.

If a device driver is statically compiled in the kernel, its registration is performed during the kernel initialization phase. Conversely, if a device driver is compiled as a kernel module (see Appendix B), its registration is performed when the module is loaded. In the latter case, the device driver can also unregister itself when the module is unloaded.

Let us consider, for instance, a generic PCI device. To properly handle it, its device driver must allocate a descriptor of type pci_driver, which is used by the PCI kernel layer to handle the device. After having initialized some fields of this descriptor, the device driver invokes the pci_register_driver( ) function. Actually, the pci_driver descriptor includes an embedded device_driver descriptor (see the earlier section "Components of the Device Driver Model"); the pci_register_function( ) simply initializes the fields of the embedded driver descriptor and invokes driver_register( ) to insert the driver in the data structures of the device driver model.

When a device driver is being registered, the kernel looks for unsupported hardware devices that could be possibly handled by the driver. To do this, it relies on the match method of the relevant bus_type bus type descriptor, and on the probe method of the device_driver object. If a hardware device that can be handled by the driver is discovered, the kernel allocates a device object and invokes device_register( ) to insert the device in the device driver model.

13.4.2. Device Driver Initialization

Registering a device driver and initializing it are two different things. A device driver is registered as soon as possible, so User Mode applications can use it through the corresponding device files. In contrast, a device driver is initialized at the last possible moment. In fact, initializing a driver means allocating precious resources of the system, which are therefore not available to other drivers.

We already have seen an example in the section "I/O Interrupt Handling" in Chapter 4: the assignment of

13.5. Character Device Drivers

Handling a character device is relatively easy, because usually sophisticated buffering strategies are not needed and disk caches are not involved. Of course, character devices differ in their requirements: some of them must implement a sophisticated communication protocol to drive the hardware device, while others just have to read a few values from a couple of I/O ports of the hardware devices. For instance, the device driver of a multiport serial card device (a hardware device offering many serial ports) is much more complicated than the device driver of a bus mouse.

Block device drivers, on the other hand, are inherently more complex than character device drivers . In fact, applications are entitled to ask repeatedly to read or write the same block of data. Furthermore, accesses to these devices are usually very slow. These peculiarities have a profound impact on the structure of the disk drivers. As we 'll see in the next chapters, however, the kernel provides sophisticated componentssuch as the page cache and the block I/O subsystemto handle them. In the rest of this chapter we focus our attention on the character device drivers.

A character device driver is described by a cdev structure, whose fields are listed in Table 13-8.

Table 13-8. The fields of the cdev structure

Type	Field	Description
struct kobject	kobj	Embedded kobject
struct module *	owner	Pointer to the module implementing the driver, if any
struct file_operations *	ops	Pointer to the file operations table of the device driver
struct list_head	list	Head of the list of inodes relative to device files for this character device
dev_t	dev	Initial major and minor numbers assigned to the device driver
unsigned int	count	Size of the range of device numbers assigned to the device driver

The list field is the head of a doubly linked circular list collecting inodes of character device files that refer to the same character device driver. There could be many device files having the same device number, and all of them refer to the same character device. Moreover, a device driver can be associated with a range of device numbers, not just a single one; all device files whose numbers fall in the range are handled by the same character device driver. The size of the range is stored in the count field.

The cdev_alloc( ) function allocates dynamically a cdev descriptor and initializes the embedded kobject so that the descriptor is automatically freed when the reference counter becomes zero.

Chapter 14. Block Device Drivers

This chapter deals with I/O drivers for block devices, i.e., for disks of every kind. The key aspect of a block device is the disparity between the time taken by the CPU and buses to read or write data and the speed of the disk hardware. Block devices have very high average access times. Each operation requires several milliseconds to complete, mainly because the disk controller must move the heads on the disk surface to reach the exact position where the data is recorded. However, when the heads are correctly placed, data transfer can be sustained at rates of tens of megabytes per second.

The organization of Linux block device handlers is quite involved. We won't be able to discuss in detail all the functions that are included in the block I/O subsystem of the kernel; however, we'll outline the general software architecture. As in the previous chapter, our objective is to explain how Linux supports the implementation of block device drivers , rather than showing how to implement one of them.

We start in the first section "Block Devices Handling" to explain the general architecture of the Linux block I/O subsystem. In the sections "The Generic Block Layer," "The I/O Scheduler," and "Block Device Drivers," we will describe the main components of the block I/O subsystem. Finally, in the last section, "Opening a Block Device File," we will outline the steps performed by the kernel when opening a block device file.

14.1. Block Devices Handling

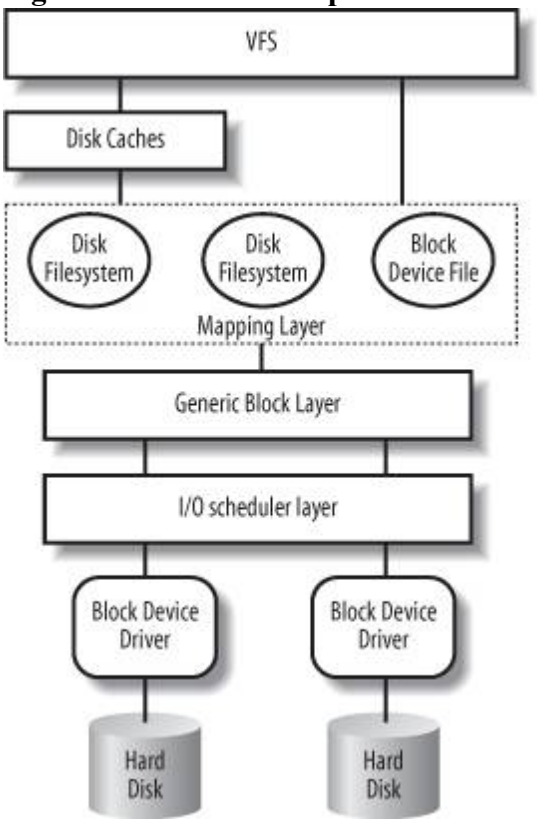
Each operation on a block device driver involves a large number of kernel components; the most important ones are shown in Figure 14-1.

Let us suppose, for instance, that a process issued a read( ) system call on some disk filewe'll see that write requests are handled essentially in the same way. Here is what the kernel typically does to service the process request:

1.

1. The service routine of the read( ) system call activates a suitable VFS function, passing to it a file descriptor and an offset inside the file. The Virtual Filesystem

1. **Figure 14-1. Kernel components affected by a block device operation**



1.

1. is the upper layer of the block device handling architecture, and it provides a common file model adopted by all filesystems supported by Linux. We have described at length the VFS layer in Chapter 12.

2.

2. The VFS function determines if the requested data is already available and, if necessary, how to perform the read operation. Sometimes there is no need to access the data on disk, because the kernel keeps in RAM the data most recently read fromor written toa block device. The disk cache mechanism is explained in Chapter 15, while details on how the VFS handles the disk operations and how it interfaces with the disk cache and the filesystems are given in Chapter 16.

3.

3. Let's assume that the kernel must read the data from the block device, thus it must determine the physical location of that data. To do this, the kernel relies on the mapping layer , which typically executes two steps:

a.

14.2. The Generic Block Layer

The generic block layer is a kernel component that handles the requests for all block devices in the system. Thanks to its functions, the kernel may easily:

-
- Put data buffers in high memorythe page frame(s) will be mapped in the kernel linear address space only when the CPU must access the data, and will be unmapped right after.
-
- Implementwith some additional efforta "zero-copy" schema, where disk data is directly put in the User Mode address space without being copied to kernel memory first; essentially, the buffer used by the kernel for the I/O transfer lies in a page frame mapped in the User Mode linear address space of a process.
-
- Manage logical volumessuch as those used by LVM (the Logical Volume Manager) and RAID (Redundant Array of Inexpensive Disks): several disk partitions, even on different block devices, can be seen as a single partition.
-
- Exploit the advanced features of the most recent disk controllers, such as large onboard disk caches , enhanced DMA capabilities, onboard scheduling of the I/O transfer requests, and so on.

14.2.1. The Bio Structure

The core data structure of the generic block layer is a descriptor of an ongoing I/O block device operation called bio. Each bio essentially includes an identifier for a disk storage areathe initial sector number and the number of sectors included in the storage areaand one or more segments describing the memory areas involved in the I/O operation. A bio is implemented by the bio data structure, whose fields are listed in Table 14-1.

Table 14-1. The fields of the bio structure

Type	Field	Description
sector_t	bi_sector	First sector on disk of block I/O operation
struct bio *	bi_next	Link to the next bio in the request queue
struct block_device *	bi_bdev	Pointer to block device descriptor
unsigned long	bi_flags	Bio status flags
unsigned long	bi_rw	I/O operation flags
		Number of segments in the bio's

14.3. The I/O Scheduler

Although block device drivers are able to transfer a single sector at a time, the block I/O layer does not perform an individual I/O operation for each sector to be accessed on disk; this would lead to poor disk performance, because locating the physical position of a sector on the disk surface is quite time-consuming. Instead, the kernel tries, whenever possible, to cluster several sectors and handle them as a whole, thus reducing the average number of head movements.

When a kernel component wishes to read or write some disk data, it actually creates a block device request. That request essentially describes the requested sectors and the kind of operation to be performed on them (read or write). However, the kernel does not satisfy a request as soon as it is createdthe I/O operation is just scheduled and will be performed at a later time. This artificial delay is paradoxically the crucial mechanism for boosting the performance of block devices. When a new block data transfer is requested, the kernel checks whether it can be satisfied by slightly enlarging a previous request that is still waiting (i.e., whether the new request can be satisfied without further seek operations). Because disks tend to be accessed sequentially, this simple mechanism is very effective.

Deferring requests complicates block device handling. For instance, suppose a process opens a regular file and, consequently, a filesystem driver wants to read the corresponding inode from disk. The block device driver puts the request on a queue, and the process is suspended until the block storing the inode is transferred. However, the block device driver itself cannot be blocked, because any other process trying to access the same disk would be blocked as well.

To keep the block device driver from being suspended, each I/O operation is processed asynchronously. In particular, block device drivers are interrupt-driven (see the section "Monitoring I/O Operations" in the previous chapter): the generic block layer invokes the I/O scheduler to create a new block device request or to enlarge an already existing one and then terminates. The block device driver, which is activated at a later time, invokes the strategy routine to select a pending request and satisfy it by issuing suitable commands to the disk controller. When the I/O operation terminates, the disk controller raises an interrupt and the corresponding handler invokes the strategy routine again, if necessary, to process another pending request.

Each block device driver maintains its own request queue, which contains the list of pending requests for the device. If the disk controller is handling several disks, there is usually one request queue for each physical block device. I/O scheduling is performed separately on each request queue, thus increasing disk performance.

14.3.1. Request Queue Descriptors

Each request queue is represented by means of a large request_queue data structure whose fields are listed in Table 14-6.

Table 14-6. The fields of the request queue descriptor

Type	Field	Description
struct list_head	queue_head	List of pending requests
struct request *	last_merge	Pointer to descriptor of the request in the queue to be considered first for possible merging

14.4. Block Device Drivers

Block device drivers are the lowest component of the Linux block subsystem. They get requests from I/O scheduler, and do whatever is required to process them.

Block device drivers are, of course, integrated within the device driver model described in the section "The Device Driver Model" in Chapter 13. Therefore, each of them refers to a device_driver descriptor; moreover, each disk handled by the driver is associated with a device descriptor. These descriptors, however, are rather generic: the block I/O subsystem must store additional information for each block device in the system.

14.4.1. Block Devices

A block device driver may handle several block devices. For instance, the IDE device driver can handle several IDE disks, each of which is a separate block device. Furthermore, each disk is usually partitioned, and each partition can be seen as a logical block device. Clearly, the block device driver must take care of all VFS system calls issued on the block device files associated with the corresponding block devices.

Each block device is represented by a block_device descriptor, whose fields are listed in Table 14-9.

Table 14-9. The fields of the block device descriptor

Type	Field	Description
dev_t	bd_dev	Major and minor numbers of the block device
struct inode *	bd_inode	Pointer to the inode of the file associated with the block device in the bdev filesystem
int	bd_openers	Counter of how many times the block device has been opened
struct semaphore	bd_sem	Semaphore protecting the opening and closing of the block device
struct semaphore	bd_mount_sem	Semaphore used to forbid new mounts on the block device
struct list_head	bd_inodes	Head of a list of inodes of opened block device files for this block device
void *	bd_holder	Current holder of block device descriptor
int	bd_holders	Counter for multiple settings of

14.5. Opening a Block Device File

We conclude this chapter by describing the steps performed by the VFS when opening a block device file.

The kernel opens a block device file every time that a filesystem is mounted over a disk or partition, every time that a swap partition is activated, and every time that a User Mode process issues an open( ) system call on a block device file. In all cases, the kernel executes essentially the same operations: it looks for the block device descriptor (possibly allocating a new descriptor if the block device is not already in use), and sets up the file operation methods for the forthcoming data transfers.

In the section "VFS Handling of Device Files" in Chapter 13, we described how the dentry_open( ) function customizes the methods of the file object when a device file is opened. In this case, the f_op field of the file object is set to the address of the def_blk_fops table, whose content is shown in Table 14-10.

Table 14-10. The default block device file operations (def_blk_fops table)

Method	Function
open	blkdev_open( )
release	blkdev_close( )
llseek	block_llseek( )
read	generic_file_read( )
write	blkdev_file_write( )
aio_read	generic_file_aio_read( )
aio_write	blkdev_file_aio_write( )
mmap	generic_file_mmap( )
fsync	block_fsync( )
ioctl	block_ioctl( )
compat-ioctl	compat_blkdev_ioctl( )
readv	generic_file_readv( )
writev	generic_file_write_nolock( )
sendfile	generic_file_sendfile( )

Chapter 15. The Page Cache

As already mentioned in the section "The Common File Model" in Chapter 12, a disk cache is a software mechanism that allows the system to keep in RAM some data that is normally stored on a disk, so that further accesses to that data can be satisfied quickly without accessing the disk.

Disk caches are crucial for system performance, because repeated accesses to the same disk data are quite common. A User Mode process that interacts with a disk is entitled to ask repeatedly to read or write the same disk data. Moreover, different processes may also need to address the same disk data at different times. As an example, you may use the *cp* command to copy a text file and then invoke your favorite editor to modify it. To satisfy your requests, the command shell will create two different processes that access the same file at different times.

We have already encountered other disk caches in Chapter 12: the dentry cache , which stores dentry objects representing filesystem pathnames, and the inode cache , which stores inode objects representing disk inodes. Notice, however, that dentry objects and inode objects are not mere buffers storing the contents of some disk blocks; thus, the dentry cache and the inode cache are rather peculiar as disk caches.

This chapter deals with the page cache , which is a disk cache working on whole pages of data. We introduce the page cache in the first section. Then, we discuss in the section "Storing Blocks in the Page Cache" how the page cache can be used to retrieve single blocks of data (for instance, superblocks and inodes); this feature is crucial to speed up the VFS and the disk-based filesystems. Next, we describe in the section "Writing Dirty Pages to Disk" how the dirty pages in the page cache are written back to disk. Finally, we mention in the last section "The sync( ), fsync( ), and fdatasync( ) System Calls" some system calls that allow a user to flush the contents of the page cache so as to update the disk contents.

15.1. The Page Cache

The page cache is the main disk cache used by the Linux kernel. In most cases, the kernel refers to the page cache when reading from or writing to disk. New pages are added to the page cache to satisfy User Mode processes's read requests. If the page is not already in the cache, a new entry is added to the cache and filled with the data read from the disk. If there is enough free memory, the page is kept in the cache for an indefinite period of time and can then be reused by other processes without accessing the disk.

Similarly, before writing a page of data to a block device, the kernel verifies whether the corresponding page is already included in the cache; if not, a new entry is added to the cache and filled with the data to be written on disk. The I/O data transfer does not start immediately: the disk update is delayed for a few seconds, thus giving a chance to the processes to further modify the data to be written (in other words, the kernel implements deferred write operations).

Kernel code and kernel data structures don't need to be read from or written to disk.[*] Thus, the pages included in the page cache can be of the following types:

[*] Well, almost never: if you want to resume the whole state of the system after a shutdown, you can perform a "suspend to disk" operation (hibernation), which saves the content of the whole RAM on a swap partition. We won't further discuss this case.

- Pages containing data of regular files; in Chapter 16, we describe how the kernel handles read, write, and memory mapping operations on them.
- Pages containing directories; as we'll see in Chapter 18, Linux handles the directories much like regular files.
- Pages containing data directly read from block device files (skipping the filesystem layer); as discussed in Chapter 16, the kernel handles them using the same set of functions as for pages containing data of regular files.
- Pages containing data of User Mode processes that have been swapped out on disk. As we'll see in Chapter 17, the kernel could be forced to keep in the page cache some pages whose contents have been already written on a swap area (either a regular file or a disk partition).
- Pages belonging to files of special filesystems, such as the shm special filesystem used for Interprocess Communication (IPC) shared memory region (see Chapter 19).

As you can see, each page included in the page cache contains data belonging to some file. This fileor more precisely the file's inodeis called the page's owner. (As we will see in Chapter 17, pages containing swapped-out data have the same owner even if they refer to different swap areas.)

Practically all read( ) and write( ) file operations rely on the page cache. The only exception occurs when a process opens a file with the O_DIRECT flag set: in this case, the page cache is bypassed and the I/O data transfers make use of buffers in the User Mode address space of the process (see the section "Direct I/O Transfers" in Chapter 16); several database applications make use of the O_DIRECT flag so that they can use their own disk caching algorithm.

15.2. Storing Blocks in the Page Cache

We have seen in the section "Block Devices Handling" in Chapter 14 that the VFS, the mapping layer, and the various filesystems group the disk data in logical units called "blocks."

In old versions of the Linux kernel, there were two different main disk caches: the page cache, which stored whole pages of disk data resulting from accesses to the contents of the disk files, and the buffer cache , which was used to keep in memory the contents of the blocks accessed by the VFS to manage the disk-based filesystems.

Starting from stable version 2.4.10, the buffer cache does not really exist anymore. In fact, for reasons of efficiency, block buffers are no longer allocated individually; instead, they are stored in dedicated pages called "buffer pages ," which are kept in the page cache.

Formally, a buffer page is a page of data associated with additional descriptors called "buffer heads ," whose main purpose is to quickly locate the disk address of each individual block in the page. In fact, the chunks of data stored in a page belonging to the page cache are not necessarily adjacent on disk.

15.2.1. Block Buffers and Buffer Heads

Each block buffer has a buffer head descriptor of type buffer_head. This descriptor contains all the information needed by the kernel to know how to handle the block; thus, before operating on each block, the kernel checks its buffer head. The fields of a buffer head are listed in Table 15-4.

Table 15-4. The fields of a buffer head

Type	Field	Description
unsigned long	b_state	Buffer status flags
struct buffer_head *	b_this_page	Pointer to the next element in the buffer page's list
struct page *	b_page	Pointer to the descriptor of the buffer page holding this block
atomic_t	b_count	Block usage counter
u32	b_size	Block size
sector_t	b_blocknr	Block number relative to the block device (logical block number)
char *	b_data	Position of the block inside the buffer page
struct block_device *	b_bdev	Pointer to block device descriptor

15.3. Writing Dirty Pages to Disk

As we have seen, the kernel keeps filling the page cache with pages containing data of block devices. Whenever a process modifies some data, the corresponding page is marked as dirtythat is, its PG_dirty flag is set.

Unix systems allow the deferred writes of dirty pages into block devices, because this noticeably improves system performance. Several write operations on a page in cache could be satisfied by just one slow physical update of the corresponding disk sectors. Moreover, write operations are less critical than read operations, because a process is usually not suspended due to delayed writings, while it is most often suspended because of delayed reads. Thanks to deferred writes, each physical block device will service, on the average, many more read requests than write ones.

A dirty page might stay in main memory until the last possible moment that is, until system shutdown. However, pushing the delayed-write strategy to its limits has two major drawbacks:

-
- If a hardware or power supply failure occurs, the contents of RAM can no longer be retrieved, so many file updates that were made since the system was booted are lost.
-
- The size of the page cache, and hence of the RAM required to contain it, would have to be hugeat least as big as the size of the accessed block devices.

Therefore, dirty pages are flushed (written) to disk under the following conditions:

-
- The page cache gets too full and more pages are needed, or the number of dirty pages becomes too large.
-
- Too much time has elapsed since a page has stayed dirty.
-
- A process requests all pending changes of a block device or of a particular file to be flushed; it does this by invoking a sync( ), fsync( ), or fdatasync( ) system call (see the section "The sync( ), fsync( ), and fdatasync( ) System Calls" later in this chapter).

Buffer pages introduce a further complication. The buffer heads associated with each buffer page allow the kernel to keep track of the status of each individual block buffer. The PG_dirty flag of the buffer page should be set if at least one of the associated buffer heads has the BH_Dirty flag set. When the kernel selects a dirty buffer page for flushing, it scans the associated buffer heads and effectively writes to disk only the contents of the dirty blocks. As soon as the kernel flushes all dirty blocks in a buffer page to disk, it clears the PG_dirty flag of the page.

15.3.1. The pdflush Kernel Threads

Earlier versions of Linux used a kernel thread called bdflush to systematically scan the page cache looking for dirty pages to flush, and they used a second kernel thread called kupdate to ensure that no page remains dirty for too long. Linux 2.6 has replaced both of them with a group of general purpose kernel threads called pdflush.

These kernel threads have a flexible structure. They act on two parameters: a pointer to a function to be

15.4. The sync( ), fsync( ), and fdatasync( ) System Calls

In this section, we examine briefly the three system calls available to user applications to flush dirty buffers to disk:

sync( )

Allows a process to flush all dirty buffers to disk

fsync( )

Allows a process to flush all blocks that belong to a specific open file to disk

fdatasync( )

Very similar to fsync( ), but doesn't flush the inode block of the file

15.4.1. The sync ( ) System Call

The service routine sys_sync( ) of the sync( ) system call invokes a series of auxiliary functions:

```
wakeup_bdflush(0);
sync_inodes(0);
sync_supers( );
sync_filesystems(0);
sync_filesystems(1);
sync_inodes(1);
```

As described in the previous section, wakeup_bdflush( ) starts a *pdflush* kernel thread, which flushes to disk all dirty pages contained in the page cache.

The sync_inodes( ) function scans the list of superblocks looking for dirty inodes to be flushed; it acts on a wait parameter that specifies whether it must wait until flushing has been performed or not. The function scans the superblocks of all currently mounted filesystems; for each superblock containing dirty inodes, sync_inodes( ) first invokes sync_sb_inodes( ) to flush the corresponding dirty pages (we described this function earlier in the section "Looking for Dirty Pages To Be Flushed"), then invokes sync_blockdev( ) to explicitly flush the dirty buffer pages owned by the block device that includes the superblock. This is done because the write_inode superblock method of many disk-based filesystems simply marks the block buffer corresponding to the disk inode as dirty; the sync_blockdev( ) function makes sure that the updates made by sync_sb_inodes( ) are effectively written to disk.

The sync_supers( ) function writes the dirty superblocks to disk, if necessary, by using the proper write_super superblock operations. Finally, the sync_filesystems( ) executes the sync_fs superblock method for all writable filesystems. This method is simply a hook offered to a filesystem in case it needs to perform some peculiar operation at each sync; this method is only used by journaling filesystems such as Ext3 (see Chapter 18).

Notice that sync_inodes( ) and sync_filesystems( ) are invoked twice, once with the wait parameter equal to 0 and the second time with the parameter equal to 1. This is done on purpose: first, they quickly flush to disk the unlocked inodes; next, they wait for each locked inode to become unlocked and finish writing them one by one.

Chapter 16. Accessing Files

Accessing a disk-based file is a complex activity that involves the VFS abstraction layer (Chapter 12), handling block devices (Chapter 14), and the use of the page cache (Chapter 15). This chapter shows how the kernel builds on all those facilities to carry out file reads and writes. The topics covered in this chapter apply both to regular files stored in disk-based filesystems and to block device files; these two kinds of files will be referred to simply as "files."

The stage we are working at in this chapter starts after the proper read or write method of a particular file has been called (as described in Chapter 12). We show here how each read ends with the desired data delivered to a User Mode process and how each write ends with data marked ready for transfer to disk. The rest of the transfer is handled by the facilities described in Chapter 14 and Chapter 15.

There are many different ways to access a file. In this chapter we will consider the following cases:

Canonical mode

The file is opened with the O_SYNC and O_DIRECT flags cleared, and its content is accessed by means of the read( ) and write( ) system calls. In this case, the read( ) system call blocks the calling process until the data is copied into the User Mode address space (however, the kernel is always allowed to return fewer bytes than requested!). The write( ) system call is different, because it terminates as soon as the data is copied into the page cache (deferred write). This case is covered in the section "Reading and Writing a File."

Synchronous mode

The file is opened with the O_SYNC flag setor the flag is set at a later time by the fcntl( ) system call. This flag affects only the write operation (read operations are always blocking), which blocks the calling process until the data is effectively written to disk. The section "Reading and Writing a File" covers this case, too.

Memory mapping mode

After opening the file, the application issues an mmap( ) system call to map the file into memory. As a result, the file appears as an array of bytes in RAM, and the application accesses directly the array elements instead of using read( ) , write( ), or lseek( ). This case is discussed in the section "Memory Mapping."

Direct I/O mode

The file is opened with the O_DIRECT flag set. Any read or write operation transfers data directly from the User Mode address space to disk, or vice versa, bypassing the page cache. We discuss this case in the section "Direct I/O Transfers." (The values of the O_SYNC and O_DIRECT flags can be combined in four meaningful ways.)

Asynchronous mode

The file is accessedeither through a group of POSIX APIs or by means of Linux-specific system callsin

16.1. Reading and Writing a File

The section "The read( ) and write( ) System Calls" in Chapter 12 described how the read( ) and write( ) system calls are implemented. The corresponding service routines end up invoking the file object's read and write methods, which may be filesystem-dependent. For disk-based filesystems, these methods locate the physical blocks that contain the data being accessed and activate the block device driver to start the data transfer.

Reading a file is page-based: the kernel always transfers whole pages of data at once. If a process issues a read( ) system call to get a few bytes, and that data is not already in RAM, the kernel allocates a new page frame, fills the page with the suitable portion of the file, adds the page to the page cache, and finally copies the requested bytes into the process address space. For most filesystems, reading a page of data from a file is just a matter of finding what blocks on disk contain the requested data. Once this is done, the kernel fills the pages by submitting the proper I/O operations to the generic block layer. In practice, the read method of all disk-based filesystems is implemented by a common function named generic_file_read( ).

Write operations on disk-based files are slightly more complicated to handle, because the file size could increase, and therefore the kernel might allocate some physical blocks on the disk. Of course, how this is precisely done depends on the filesystem type. However, many disk-based filesystems implement their write methods by means of a common function named generic_file_write( ). Examples of such filesystems are Ext2, System V /Coherent /Xenix , and MINIX . On the other hand, several other filesystems, such as journaling and network filesystems , implement the write method by means of custom functions.

16.1.1. Reading from a File

The generic_file_read( ) function is used to implement the read method for block device files and for regular files of almost all disk-based filesystems. This function acts on the following parameters:

filp

Address of the file object

buf

Linear address of the User Mode memory area where the characters read from the file must be stored

count

Number of characters to be read

ppos

Pointer to a variable that stores the offset from which reading must start (usually the f_pos field of the filp file object)

As a first step, the function initializes two descriptors. The first descriptor is stored in the local variable local_iov of type iovec; it contains the address (buf) and the length (count) of the User Mode buffer that shall receive the data read from the file. The second descriptor is stored in the local variable kiocb of type kiocb; it is used to keep track of the completion status of an ongoing synchronous or asynchronous

16.2. Memory Mapping

As already mentioned in the section "Memory Regions" in Chapter 9, a memory region can be associated with some portion of either a regular file in a disk-based filesystem or a block device file. This means that an access to a byte within a page of the memory region is translated by the kernel into an operation on the corresponding byte of the file. This technique is called memory mapping.

Two kinds of memory mapping exist:

Shared

Each write operation on the pages of the memory region changes the file on disk; moreover, if a process writes into a page of a shared memory mapping, the changes are visible to all other processes that map the same file.

Private

Meant to be used when the process creates the mapping just to read the file, not to write it. For this purpose, private mapping is more efficient than shared mapping. But each write operation on a privately mapped page will cause it to stop mapping the page in the file. Thus, a write does not change the file on disk, nor is the change visible to any other processes that access the same file. However, pages of a private memory mapping that have not been modified by the process are affected by file updates performed by other processes.

A process can create a new memory mapping by issuing an mmap( ) system call (see the section "Creating a Memory Mapping" later in this chapter). Programmers must specify either the MAP_SHARED flag or the MAP_PRIVATE flag as a parameter of the system call; as you can easily guess, in the former case the mapping is shared, while in the latter it is private. Once the mapping is created, the process can read the data stored in the file by simply reading from the memory locations of the new memory region. If the memory mapping is shared, the process can also modify the corresponding file by simply writing into the same memory locations. To destroy or shrink a memory mapping, the process may use the munmap( ) system call (see the later section "Destroying a Memory Mapping").

As a general rule, if a memory mapping is shared, the corresponding memory region has the VM_SHARED flag set; if it is private, the VM_SHARED flag is cleared. As we'll see later, an exception to this rule exists for read-only shared memory mappings.

16.2.1. Memory Mapping Data Structures

A memory mapping is represented by a combination of the following data structures :

- The inode object associated with the mapped file
- The address_space object of the mapped file
- A file object for each different mapping performed on the file by different processes

16.3. Direct I/O Transfers

As we have seen, in Version 2.6 of Linux, there is no substantial difference between accessing a regular file through the filesystem, accessing it by referencing its blocks on the underlying block device file, or even establishing a file memory mapping. There are, however, some highly sophisticated programs (self-caching applications) that would like to have full control of the whole I/O data transfer mechanism. Consider, for example, high-performance database servers: most of them implement their own caching mechanisms that exploit the peculiar nature of the queries to the database. For these kinds of programs, the kernel page cache doesn't help; on the contrary, it is detrimental for the following reasons:

-
- Lots of page frames are wasted to duplicate disk data already in RAM (in the user-level disk cache).
-
- The read( ) and write( ) system calls are slowed down by the redundant instructions that handle the page cache and the read-ahead; ditto for the paging operations related to the file memory mappings.
-
- Rather than transferring the data directly between the disk and the user memory, the read( ) and write( ) system calls make two transfers: between the disk and a kernel buffer and between the kernel buffer and the user memory.

Because block hardware devices must be handled through interrupts and Direct Memory Access (DMA), and this can be done only in Kernel Mode, some sort of kernel support is definitely required to implement self-caching applications.

Linux offers a simple way to bypass the page cache: direct I/O transfers. In each I/O direct transfer, the kernel programs the disk controller to transfer the data directly from/to pages belonging to the User Mode address space of a self-caching application.

As we know, each data transfer proceeds asynchronously. While it is in progress, the kernel may switch the current process, the CPU may return to User Mode, the pages of the process that raised the data transfer might be swapped out, and so on. This works just fine for ordinary I/O data transfers because they involve pages of the disk caches . Disk caches are owned by the kernel, cannot be swapped out, and are visible to all processes in Kernel Mode.

On the other hand, direct I/O transfers should move data within pages that belong to the User Mode address space of a given process. The kernel must take care that these pages are accessible by every process in Kernel Mode and that they are not swapped out while the data transfer is in progress. Let us see how this is achieved.

When a self-caching application wishes to directly access a file, it opens the file specifying the O_DIRECT flag (see the section "The open( ) System Call" in Chapter 12). While servicing the open( ) system call, the dentry_open( ) function checks whether the direct_IO method is implemented for the address_space object of the file being opened, and returns an error code in the opposite case. The O_DIRECT flag can also be set for a file already opened by using the F_SETFL command of the fcntl( ) system call.

Let us consider first the case where the self-caching application issues a read( ) system call on a file with O_DIRECT. As mentioned in the section "Reading from a File" earlier in this chapter, the read file method is usually implemented by the generic_file_read( ) function, which initializes the iovec and kiocb descriptors and invokes generic_file_aio_read( ). The latter function verifies that the User Mode

16.4. Asynchronous I/O

The POSIX 1003.1 standard defines a set of library functionslisted in Table 16-4for accessing the files in an asynchronous way. "Asynchronous" essentially means that when a User Mode process invokes a library function to read or write a file, the function terminates as soon as the read or write operation has been enqueued, possibly even before the actual I/O data transfer takes place. The calling process can thus continue its execution while the data is being transferred.

Table 16-4. The POSIX library functions for asynchronous I/O

Function	Description
aio_read( )	Asynchronously reads some data from a file
aio_write( )	Asynchronously writes some data into a file
aio_fsync( )	Requests a flush operation for all outstanding asynchronous I/O operations (does not block)
aio_error( )	Gets the error code for an outstanding asynchronous I/O operation
aio_return( )	Gets the return code for a completed asynchronous I/O operation
aio_cancel( )	Cancels an outstanding asynchronous I/O operation
aio_suspend( )	Suspends the process until at least one of several outstanding I/O operations completes

Using asynchronous I/O is quite simple. The application opens the file by means of the usual open( ) system call. Then, it fills up a control block of type struct aiocb with the information describing the requested operation. The most commonly used fields of the struct aiocb control block are:

aio_fildes

The file descriptor of the file (as returned by the open( ) system call)

aio_buf

The User Mode buffer for the file's data

aio_nbytes

How many bytes should be transferred

Chapter 17. Page Frame Reclaiming

In previous chapters, we explained how the kernel handles dynamic memory by keeping track of free and busy page frames. We have also discussed how every process in User Mode has its own address space and has its requests for memory satisfied by the kernel one page at a time, so that page frames can be assigned to the process at the very last possible moment. Last but not least, we have shown how the kernel makes use of dynamic memory to implement both memory and disk caches .

In this chapter, we complete our description of the virtual memory subsystem by discussing page frame reclaiming. We'll start in the first section, "The Page Frame Reclaiming Algorithm," explaining why the kernel needs to reclaim page frames and what strategy it uses to achieve this. We then make a technical digression in the section "Reverse Mapping" to discuss the data structures used by the kernel to locate quickly all the Page Table entries that point to the same page frame. The section "Implementing the PFRA" is devoted to the page frame reclaiming algorithm used by Linux. The last main section, "Swapping," is almost a chapter by itself: it covers the swap subsystem, a kernel component used to save anonymous (not mapping data of files) pages on disk.

PREV NEXT

17.1. The Page Frame Reclaiming Algorithm

One of the fascinating aspects of Linux is that the checks performed before allocating dynamic memory to User Mode processes or to the kernel are somewhat perfunctory.

No rigorous check is made, for instance, on the total amount of RAM assigned to the processes created by a single user (the limits mentioned in the section "Process Resource Limits" in Chapter 3 mostly affect single processes). Similarly, no limit is placed on the size of the many disk caches and memory caches used by the kernel.

This lack of controls is a design choice that allows the kernel to use the available RAM in the best possible way. When the system load is low, the RAM is filled mostly by the disk caches and the few running processes can benefit from the information stored in them. However, when the system load increases, the RAM is filled mostly by pages of the processes and the caches are shrunken to make room for additional processes.

As we saw in previous chapters, both memory and disk caches grab more and more page frames but never release any of them. This is reasonable because cache systems don't know if and when processes will reuse some of the cached data and are therefore unable to identify the portions of cache that should be released. Moreover, thanks to the demand paging mechanism described in Chapter 9, User Mode processes get page frames as long as they proceed with their execution; however, demand paging has no way to force processes to release the page frames whenever they are no longer used.

Thus, sooner or later all the free memory will be assigned to processes and caches. The page frame reclaiming algorithm of the Linux kernel refills the lists of free blocks of the buddy system by "stealing" page frames from both User Mode processes and kernel caches.

Actually, page frame reclaiming must be performed before all the free memory has been used up. Otherwise, the kernel might be easily trapped in a deadly chain of memory requests that leads to a system crash. Essentially, to free a page frame the kernel must write its data to disk; however, to accomplish this operation, the kernel requires another page frame (for instance, to allocate the buffer heads for the I/O data transfer). If no free page frame exists, no page frame can be freed.

One of the goals of page frame reclaiming is thus to conserve a minimal pool of free page frames so that the kernel may safely recover from "low on memory" conditions.

17.1.1. Selecting a Target Page

The objective of the page frame reclaiming algorithm (PFRA) is to pick up page frames and make them free. Clearly the page frames selected by the PFRA must be non-free , that is, they must not be already included in one of the free_area arrays used by the buddy system (see the section "The Buddy System Algorithm" in Chapter 8).

The PFRA handles the page frames in different ways, according to their contents. We can distinguish between unreclaimable pages, swappable pages, syncable pages, and discardable pages. These types are explained in Table 17-1.

Table 17-1. The types of pages considered by the PFRA

Type of pages	Description	Reclaim action
	Free pages (included in buddy system lists)	

17.2. Reverse Mapping

As stated in the previous section, one of the objectives of the PFRA is to be able to free a shared page frame. To that end, the Linux 2.6 kernel is able to locate quickly all the Page Table entries that point to the same page frame. This activity is called reverse mapping .

A trivial solution for reverse mapping would be to include in each page descriptor additional fields to link together all the Page Table entries that point to the page frame associated with the page descriptor. However, keeping such lists up-to-date would increase significantly the kernel overhead; for that reason, more sophisticated solutions have been devised. The technique used in Linux 2.6 is named object-based reverse mapping. Essentially, for any reclaimable User Mode page, the kernel stores the backward links to all memory regions in the system (the "objects") that include the page itself. Each memory region descriptor stores a pointer to a memory descriptor, which in turn includes a pointer to a Page Global Directory. Therefore, the backward links enable the PFRA to retrieve all Page Table entries referencing a given a page. Because there are fewer memory region descriptors than page descriptors, updating the backward links of a shared page is less time consuming. Let's see how this scheme is worked out.

First of all, the PFRA must have a way to determine whether the page to be reclaimed is shared or non-shared, and whether it is mapped or anonymous. In order to do this, the kernel looks at two fields of the page descriptor: _mapcount and mapping.

The _mapcount field stores the number of Page Table entries that refer to the page frame. The counter starts from -1: this value means that no Page Table entry references the page frame. Thus, if the counter is zero, the page is non-shared, while if it is greater than zero the page is shared. The page_mapcount( ) function receives the address of a page descriptor and returns the value of its _mapcount plus one (thus, for instance, it returns one for a non-shared page included in the User Mode address space of some process).

The mapping field of the page descriptor determines whether the page is mapped or anonymous, as follows:

- If the mapping field is NULL, the page belongs to the swap cache (see the section "The Swap Cache" later in this chapter).
- If the mapping field is not NULL and its least significant bit is 1, it means the page is anonymous and the mapping field encodes the pointer to an anon_vma descriptor (see the next section, " Reverse Mapping for Anonymous Pages").
- If the mapping field is non-NULL and its least significant bit is 0, the page is mapped; the mapping field points to the address_space object of the corresponding file (see the section "The address_space Object" in Chapter 15).

Every address_space object used by Linux is aligned in RAM so that its starting linear address is a multiple of four. Therefore, the least significant bit of the mapping field can be used as a flag denoting whether the field contains a pointer to an address_space object or to an anon_vma descriptor. This is a dirty programming trick, but the kernel uses a lot of page descriptors, thus these data structures should be as small as possible. The PageAnon( ) function receives as its parameter the address of a page descriptor and returns 1 if the least significant bit of the mapping field is set, 0 otherwise.

The TRy_to_unmap( ) function, which receives as its parameter a pointer to a page descriptor, tries to

17.3. Implementing the PFRA

The page frame reclaiming algorithm must take care of many kinds of pages owned by User Mode processes, disk caches and memory caches; moreover, it has to obey several heuristic rules. Thus, it is not surprising that the PFRA is composed of a large number of functions. Figure 17-3 shows the main PFRA functions; an arrow denotes a function invocation, thus for instance try_to_free_pages( ) invokes shrink_caches( ), shrink_slab( ), and out_of_memory( ).

As you can see, there are several "entry points" for the PFRA. Actually, page frame reclaiming is performed on essentially three occasions:

Low on memory reclaiming

The kernel detects a "low on memory" condition.

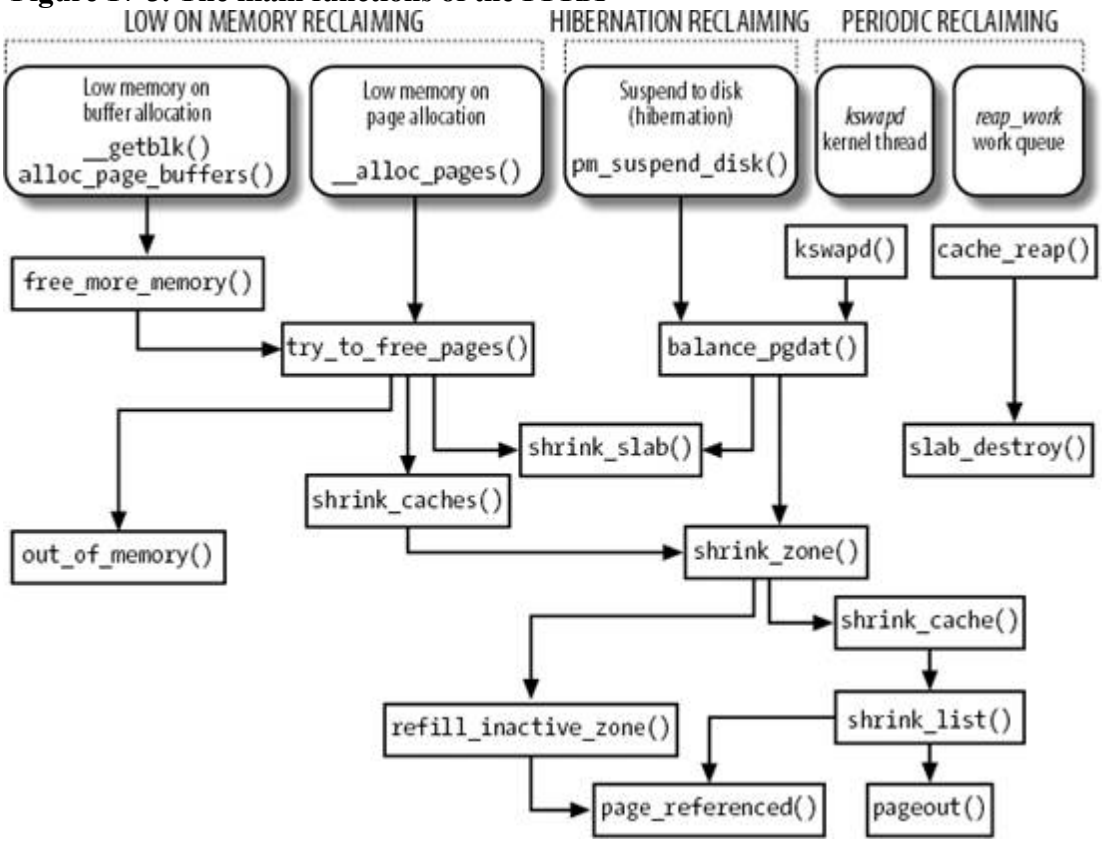
Hibernation reclaiming

The kernel must free memory because it is entering in the suspend-to-disk state (we don't further discuss this case).

Periodic reclaiming

A kernel thread is activated periodically to perform memory reclaiming, if necessary.

Figure 17-3. The main functions of the PFRA



Low on memory reclaiming is activated in the following cases:

17.4. Swapping

Swapping has been introduced to offer a backup on disk for unmapped pages. We know from the previous discussion that there are three kinds of pages that must be handled by the swapping subsystem:

- Pages that belong to an anonymous memory region of a process (User Mode stack or heap)
- Dirty pages that belong to a private memory mapping of a process
- Pages that belong to an IPC shared memory region (see the section "IPC Shared Memory" in Chapter 19)

Like demand paging, swapping must be transparent to programs. In other words, no special instruction related to swapping needs to be inserted into the code. To understand how this can be done, recall from the section "Regular Paging" in Chapter 2 that each Page Table entry includes a Present flag. The kernel exploits this flag to signal that a page belonging to a process address space has been swapped out. Besides that flag, Linux also takes advantage of the remaining bits of the Page Table entry to store into them a "swapped-out page identifier" that encodes the location of the swapped-out page on disk. When a Page Fault exception occurs, the corresponding exception handler can detect that the page is not present in RAM and invoke the function that swaps in the missing page from disk.

The main features of the swapping subsystem can be summarized as follows:

- Set up "swap areas" on disk to store pages that do not have a disk image.
- Manage the space on swap areas allocating and freeing "page slots" as the need occurs.
- Provide functions both to "swap out" pages from RAM into a swap area and to "swap in" pages from a swap area into RAM.
- Make use of "swapped-out page identifiers" in the Page Table entries of pages that are currently swapped out to keep track of the positions of data in the swap areas.

To sum up, swapping is the crowning feature of page frame reclaiming. If we want to be sure that all the page frames obtained by a process, and not only those containing pages that have an image on disk, can be reclaimed at will by the PFRA, then swapping has to be used. Of course, you might turn off swapping by using the *swapoff* command; in this case, however, disk thrashing is likely to occur sooner when the system load increases.

We should also mention that swapping can be used to expand the memory address space that is effectively usable by the User Mode processes. In fact, large swap areas allow the kernel to launch several demanding applications whose total memory requests exceed the amount of physical RAM installed in the system. However, simulation of RAM is not like RAM in terms of performance. Every access by a process to a page that is currently swapped out is of several orders of magnitude longer than an access to a page in RAM. In short, if performance is of great importance, swapping should be used only as a last resort; adding RAM chips still remains the best solution to cope with increasing computing

Chapter 18. The Ext2 and Ext3 Filesystems

In this chapter, we finish our extensive discussion of I/O and filesystems by taking a look at the details the kernel has to take care of when interacting with a specific filesystem. Because the Second Extended Filesystem (Ext2) is native to Linux and is used on virtually every Linux system, it is a natural choice for this discussion. Furthermore, Ext2 illustrates a lot of good practices in its support for modern filesystem features with fast performance. To be sure, other filesystems supported by Linux include many interesting features, but we have no room to examine all of them.

After introducing Ext2 in the section "General Characteristics of Ext2," we describe the data structures needed, just as in other chapters. Because we are looking at a specific way to store data on disk, we have to consider two versions of the same data structures. The section "Ext2 Disk Data Structures" shows the data structures stored by Ext2 on disk, while "Ext2 Memory Data Structures" shows the corresponding versions in memory.

Then we get to the operations performed on the filesystem. In the section "Creating the Ext2 Filesystem," we discuss how Ext2 is created in a disk partition. The next sections describe the kernel activities performed whenever the disk is used. Most of these are relatively low-level activities dealing with the allocation of disk space to inodes and data blocks.

In the last section, we give a short description of the Ext3 filesystem, which is the next step in the evolution of the Ext2 filesystem .

18.1. General Characteristics of Ext2

Unix-like operating systems use several types of filesystems. Although the files of all such filesystems have a common subset of attributes required by a few POSIX APIs such as stat( ), each filesystem is implemented in a different way.

The first versions of Linux were based on the MINIX filesystem. As Linux matured, the Extended Filesystem (Ext FS) was introduced; it included several significant extensions, but offered unsatisfactory performance. The Second Extended Filesystem (Ext2) was introduced in 1994; besides including several new features , it is quite efficient and robust and is, together with its offspring Ext3, the most widely used Linux filesystem.

The following features contribute to the efficiency of Ext2:

- When creating an Ext2 filesystem, the system administrator may choose the optimal block size (from 1,024 to 4,096 bytes), depending on the expected average file length. For instance, a 1,024-block size is preferable when the average file length is smaller than a few thousand bytes because this leads to less internal fragmentationthat is, less of a mismatch between the file length and the portion of the disk that stores it (see the section "Memory Area Management" in Chapter 8, where internal fragmentation for dynamic memory was discussed). On the other hand, larger block sizes are usually preferable for files greater than a few thousand bytes because this leads to fewer disk transfers, thus reducing system overhead.
- When creating an Ext2 filesystem, the system administrator may choose how many inodes to allow for a partition of a given size, depending on the expected number of files to be stored on it. This maximizes the effectively usable disk space.
- The filesystem partitions disk blocks into groups. Each group includes data blocks and inodes stored in adjacent tracks. Thanks to this structure, files stored in a single block group can be accessed with a lower average disk seek time.
- The filesystem preallocates disk data blocks to regular files before they are actually used. Thus, when the file increases in size, several blocks are already reserved at physically adjacent positions, reducing file fragmentation.
- Fast symbolic links (see the section "Hard and Soft Links" in Chapter 1) are supported. If the symbolic link represents a short pathname (at most 60 characters), it can be stored in the inode and can thus be translated without reading a data block.

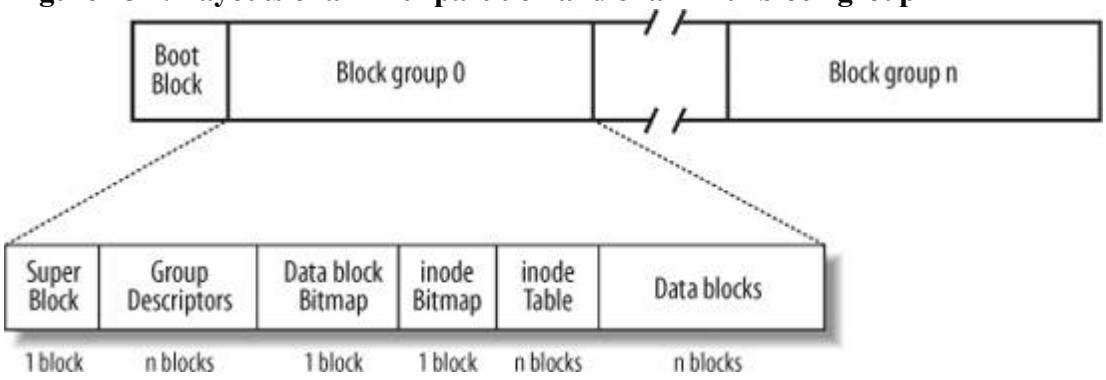
Moreover, the Second Extended Filesystem includes other features that make it both robust and flexible:

- A careful implementation of file-updating that minimizes the impact of system crashes. For instance, when creating a new hard link for a file, the counter of hard links in the disk inode is increased first, and the new name is added into the proper directory next. In this way, if a hardware failure occurs after the inode update but before the directory can be changed, the directory is consistent, even if the inode's hard link counter is wrong. Deleting the file does not lead to catastrophic results, although the file's data blocks cannot be automatically reclaimed. If

18.2. Ext2 Disk Data Structures

The first block in each Ext2 partition is never managed by the Ext2 filesystem, because it is reserved for the partition boot sector (see Appendix A). The rest of the Ext2 partition is split into block groups , each of which has the layout shown in Figure 18-1. As you will notice from the figure, some data structures must fit in exactly one block, while others may require more than one block. All the block groups in the filesystem have the same size and are stored sequentially, thus the kernel can derive the location of a block group in a disk simply from its integer index.

Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group



Block groups reduce file fragmentation, because the kernel tries to keep the data blocks belonging to a file in the same block group, if possible. Each block in a block group contains one of the following pieces of information:

- A copy of the filesystem's superblock
- A copy of the group of block group descriptors
- A data block bitmap
- An inode bitmap
- A table of inodes
- A chunk of data that belongs to a file; i.e., data blocks

If a block does not contain any meaningful information, it is said to be free.

As you can see from Figure 18-1, both the superblock and the group descriptors are duplicated in each block group. Only the superblock and the group descriptors included in block group 0 are used by the kernel, while the remaining superblocks and group descriptors are left unchanged; in fact, the kernel doesn't even look at them. When the *e2fsck* program executes a consistency check on the filesystem status, it refers to the superblock and the group descriptors stored in block group 0, and then copies them into all other block groups. If data corruption occurs and the main superblock or the main group descriptors in block group 0 become invalid, the system administrator can instruct *e2fsck* to refer to the

18.3. Ext2 Memory Data Structures

For the sake of efficiency, most information stored in the disk data structures of an Ext2 partition are copied into RAM when the filesystem is mounted, thus allowing the kernel to avoid many subsequent disk read operations. To get an idea of how often some data structures change, consider some fundamental operations:

-
- When a new file is created, the values of the s_free_inodes_count field in the Ext2 superblock and of the bg_free_inodes_count field in the proper group descriptor must be decreased.
-
- If the kernel appends some data to an existing file so that the number of data blocks allocated for it increases, the values of the s_free_blocks_count field in the Ext2 superblock and of the bg_free_blocks_count field in the group descriptor must be modified.
-
- Even just rewriting a portion of an existing file involves an update of the s_wtime field of the Ext2 superblock.

Because all Ext2 disk data structures are stored in blocks of the Ext2 partition, the kernel uses the page cache to keep them up-to-date (see the section "Writing Dirty Pages to Disk" in Chapter 15).

Table 18-6 specifies, for each type of data related to Ext2 filesystems and files, the data structure used on the disk to represent its data, the data structure used by the kernel in memory, and a rule of thumb used to determine how much caching is used. Data that is updated very frequently is always cached; that is, the data is permanently stored in memory and included in the page cache until the corresponding Ext2 partition is unmounted. The kernel gets this result by keeping the page's usage counter greater than 0 at all times.

Table 18-6. VFS images of Ext2 data structures

Type	Disk data structure	Memory data structure	Caching mode
Superblock	ext2_super_block	ext2_sb_info	Always cached
Group descriptor	ext2_group_desc	ext2_group_desc	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	ext2_inode	ext2_inode_info	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	ext2_inode	None	Never
Free block	Array of bytes	None	Never

18.4. Creating the Ext2 Filesystem

There are generally two stages to creating a filesystem on a disk. The first step is to format it so that the disk driver can read and write blocks on it. Modern hard disks come preformatted from the factory and need not be reformatted; floppy disks may be formatted on Linux using a utility program such as *superformat* or *fdformat*. The second step involves creating a filesystem, which means setting up the structures described in detail earlier in this chapter.

Ext2 filesystems are created by the *mke2fs* utility program; it assumes the following default options, which may be modified by the user with flags on the command line:

- Block size: 1,024 bytes (default value for a small filesystem)
- Fragment size: block size (block fragmentation is not implemented)
- Number of allocated inodes: 1 inode for each 8,192 bytes
- Percentage of reserved blocks: 5 percent

The program performs the following actions:

1. Initializes the superblock and the group descriptors.
2. Optionally, checks whether the partition contains defective blocks; if so, it creates a list of defective blocks.
3. For each block group, reserves all the disk blocks needed to store the superblock, the group descriptors, the inode table, and the two bitmaps.
4. Initializes the inode bitmap and the data map bitmap of each block group to 0.
5. Initializes the inode table of each block group.
6. Creates the */root* directory.
7. Creates the *lost+found* directory, which is used by *e2fsck* to link the lost and found defective blocks.
8. Updates the inode bitmap and the data block bitmap of the block group in which the two previous directories have been created.
9.

18.5. Ext2 Methods

Many of the VFS methods described in Chapter 12 have a corresponding Ext2 implementation. Because it would take a whole book to describe all of them, we limit ourselves to briefly reviewing the methods implemented in Ext2. Once the disk and the memory data structures are clearly understood, the reader should be able to follow the code of the Ext2 functions that implement them.

18.5.1. Ext2 Superblock Operations

Many VFS superblock operations have a specific implementation in Ext2, namely alloc_inode, destroy_inode, read_inode, write_inode, delete_inode, put_super, write_super, statfs, remount_fs, and clear_inode. The addresses of the superblock methods are stored in the ext2_sops array of pointers.

18.5.2. Ext2 inode Operations

Some of the VFS inode operations have a specific implementation in Ext2, which depends on the type of the file to which the inode refers.

The inode operations for Ext2 regular files and Ext2 directories are shown in Table 18-8; the purpose of each method is described in the section "Inode Objects" in Chapter 12. The table does not show the methods that are undefined (a NULL pointer) for both regular files and directories; recall that if a method is undefined, the VFS either invokes a generic function or does nothing at all. The addresses of the Ext2 methods for regular files and directories are stored in the ext2_file_inode_operations and ext2_dir_inode_operations tables, respectively.

Table 18-8. Ext2 inode operations for regular files and directories

VFS inode operation	Regular file	Directory
create	NULL	ext2_create( )
lookup	NULL	ext2_lookup( )
link	NULL	ext2_link( )
unlink	NULL	ext2_unlink( )
symlink	NULL	ext2_symlink( )
mkdir	NULL	ext2_mkdir( )
rmdir	NULL	ext2_rmdir( )
mknod	NULL	ext2_mknod( )
rename	NULL	ext2_rename( )
truncate	ext2_TRuncate( )	NULL

18.6. Managing Ext2 Disk Space

The storage of a file on disk differs from the view the programmer has of the file in two ways: blocks can be scattered around the disk (although the filesystem tries hard to keep blocks sequential to improve access time), and files may appear to a programmer to be bigger than they really are because a program can introduce holes into them (through the lseek( ) system call).

In this section, we explain how the Ext2 filesystem manages the disk space how it allocates and deallocates inodes and data blocks. Two main problems must be addressed:

-
- Space management must make every effort to avoid file fragmentation the physical storage of a file in several, small pieces located in non-adjacent disk blocks. File fragmentation increases the average time of sequential read operations on the files, because the disk heads must be frequently repositioned during the read operation.[*] This problem is similar to the external fragmentation of RAM discussed in the section "The Buddy System Algorithm" in Chapter 8.
- [*] Please note that fragmenting a file across block groups (A Bad Thing) is quite different from the not-yet-implemented fragmentation of blocks to store many files in one block (A Good Thing).
-
- Space management must be time-efficient; that is, the kernel should be able to quickly derive from a file offset the corresponding logical block number in the Ext2 partition. In doing so, the kernel should limit as much as possible the number of accesses to addressing tables stored on disk, because each such intermediate access considerably increases the average file access time.

18.6.1. Creating inodes

The ext2_new_inode( ) function creates an Ext2 disk inode, returning the address of the corresponding inode object (or NULL, in case of failure). The function carefully selects the block group that contains the new inode; this is done to spread unrelated directories among different groups and, at the same time, to put files into the same group as their parent directories. To balance the number of regular files and directories in a block group, Ext2 introduces a "debt" parameter for every block group.

The function acts on two parameters: the address dir of the inode object that refers to the directory into which the new inode must be inserted and a mode that indicates the type of inode being created. The latter argument also includes the MS_SYNCHRONOUS mount flag (see the section "Mounting a Generic Filesystem" in Chapter 12) that requires the current process to be suspended until the inode is allocated. The function performs the following actions:

1.
1. Invokes new_inode( ) to allocate a new VFS inode object; initializes its i_sb field to the superblock address stored in dir->i_sb, and adds it to the in-use inode list and to the superblock's list (see the section "Inode Objects" in Chapter 12).
2.
2. If the new inode is a directory, the function invokes find_group_orlov( ) to find a suitable block group for the directory.[*] This function implements the following heuristics:
2. [*] The Ext2 filesystem may also be mounted with an option flag that forces the kernel to make use of a simpler, older allocation strategy, which is implemented by the find_group_dir( ) function.

18.7. The Ext3 Filesystem

In this section we'll briefly describe the enhanced filesystem that has evolved from Ext2, named Ext3. The new filesystem has been designed with two simple concepts in mind:

- To be a journaling filesystem (see the next section)
- To be, as much as possible, compatible with the old Ext2 filesystem

Ext3 achieves both the goals very well. In particular, it is largely based on Ext2, so its data structures on disk are essentially identical to those of an Ext2 filesystem. As a matter of fact, if an Ext3 filesystem has been cleanly unmounted, it can be remounted as an Ext2 filesystem; conversely, creating a journal of an Ext2 filesystem and remounting it as an Ext3 filesystem is a simple, fast operation.

Thanks to the compatibility between Ext3 and Ext2, most descriptions in the previous sections of this chapter apply to Ext3 as well. Therefore, in this section, we focus on the new feature offered by Ext3 "the journal."

18.7.1. Journaling Filesystems

As disks became larger, one design choice of traditional Unix filesystems (such as Ext2) turns out to be inappropriate. As we know from Chapter 14, updates to filesystem blocks might be kept in dynamic memory for long period of time before being flushed to disk. A dramatic event such as a power-down failure or a system crash might thus leave the filesystem in an inconsistent state. To overcome this problem, each traditional Unix filesystem is checked before being mounted; if it has not been properly unmounted, then a specific program executes an exhaustive, time-consuming check and fixes all the filesystem's data structures on disk.

For instance, the Ext2 filesystem status is stored in the s_mount_state field of the superblock on disk. The *e2fsck* utility program is invoked by the boot script to check the value stored in this field; if it is not equal to EXT2_VALID_FS, the filesystem was not properly unmounted, and therefore *e2fsck* starts checking all disk data structures of the filesystem.

Clearly, the time spent checking the consistency of a filesystem depends mainly on the number of files and directories to be examined; therefore, it also depends on the disk size. Nowadays, with filesystems reaching hundreds of gigabytes, a single consistency check may take hours. The involved downtime is unacceptable for every production environment or high-availability server.

The goal of a journaling filesystem is to avoid running time-consuming consistency checks on the whole filesystem by looking instead in a special disk area that contains the most recent disk write operations named journal. Remounting a journaling filesystem after a system failure is a matter of a few seconds.

18.7.2. The Ext3 Journaling Filesystem

The idea behind Ext3 journaling is to perform each high-level change to the filesystem in two steps. First, a copy of the blocks to be written is stored in the journal; then, when the I/O data transfer to the journal is completed (in short, data is committed to the journal), the blocks are written in the filesystem. When the I/O data transfer to the filesystem terminates (data is committed to the filesystem), the copies of the blocks in the journal are discarded.

While recovering after a system failure, the *e2fsck* program distinguishes the following two cases:

Chapter 19. Process Communication

This chapter explains how User Mode processes can synchronize their actions and exchange data. We already covered several synchronization topics in Chapter 5, but the actors there were kernel control paths, not User Mode programs. We are now ready, after having discussed I/O management and filesystems at length, to extend the discussion to User Mode processes. These processes must rely on the kernel to facilitate interprocess synchronization and communication.

As we saw in the section "Linux File Locking" in Chapter 12, a form of synchronization among User Mode processes can be achieved by creating a (possibly empty) file and using suitable VFS system calls to lock and unlock it. While processes can similarly share data via temporary files protected by locks, this approach is costly because it requires accesses to the filesystem on disk. For this reason, all Unix kernels include a set of system calls that supports process communication without interacting with the filesystem; furthermore, several wrapper functions were developed and inserted in suitable libraries to expedite how processes issue their synchronization requests to the kernel.

As usual, application programmers have a variety of needs that call for different communication mechanisms. Here are the basic mechanisms that Unix systems offer to allow interprocess communication:

Pipes and FIFOs (named pipes)

Best suited to implement producer/consumer interactions among processes. Some processes fill the pipe with data, while others extract data from the pipe. They are covered in the sections "Pipes" and "FIFOs."

Semaphores

Represent, as the name implies, the User Mode version of the kernel semaphores discussed in the section "Semaphores" in Chapter 5. They are described in the section "System V IPC."

Messages

Allow processes to exchange messages (short blocks of data) by reading and writing them in predefined message queues. The Linux kernel offers two different versions of messages: System V IPC messages (covered in the section "System V IPC") and POSIX messages (described in the section "POSIX Message Queues").

Shared memory regions

Allow processes to exchange information via a shared block of memory. In applications that must share large amounts of data, this can be the most efficient form of process communication. They are described in the section "System V IPC."

Sockets

Allow processes on different computers to exchange data through a network. Sockets can also be used as a communication tool for processes located on the same host computer; the X Window System graphic interface, for instance, uses a socket to allow client programs to exchange data with the X server.

19.1. Pipes

Pipes are an interprocess communication mechanism that is provided in all flavors of Unix. A pipe is a one-way flow of data between processes: all data written by a process to the pipe is routed by the kernel to another process, which can thus read it.

In Unix command shells, pipes can be created by means of the | operator. For instance, the following statement instructs the shell to create two processes connected by a pipe:

```
$ ls | more
```

The standard output of the first process, which executes the *ls* program, is redirected to the pipe; the second process, which executes the *more* program, reads its input from the pipe.

Note that the same results can also be obtained by issuing two commands such as the following:

```
$ ls > temp
$ more < temp
```

The first command redirects the output of *ls* into a regular file; then the second command forces *more* to read its input from the same file. Of course, using pipes instead of temporary files is usually more convenient due to the following reasons:

-
- The shell statement is much shorter and simpler.
-
- There is no need to create temporary regular files, which must be deleted later.

19.1.1. Using a Pipe

Pipes may be considered open files that have no corresponding image in the mounted filesystems. A process creates a new pipe by means of the pipe( ) system call, which returns a pair of file descriptors ; the process may then pass these descriptors to its descendants through fork( ) , thus sharing the pipe with them. The processes can read from the pipe by using the read( ) system call with the first file descriptor; likewise, they can write into the pipe by using the write( ) system call with the second file descriptor.

POSIX defines only half-duplex pipes , so even though the pipe( ) system call returns two file descriptors, each process must close one before using the other. If a two-way flow of data is required, the processes must use two different pipes by invoking pipe( ) twice.

Several Unix systems, such as System V Release 4, implement full-duplex pipes . In a full-duplex pipe, both descriptors can be written into and read from, thus there are two bidirectional channels of information. Linux adopts yet another approach: each pipe's file descriptors are still one-way, but it is not necessary to close one of them before using the other.

Let's resume the previous example. When the command shell interprets the ls|more statement, it essentially performs the following actions:

1.

1. Invokes the pipe( ) system call; let's assume that pipe( ) returns the file descriptors 3 (the pipe's read channel) and 4 (the write channel).
2.

19.2. FIFOs

Although pipes are a simple, flexible, and efficient communication mechanism, they have one main drawbacknamely, that there is no way to open an already existing pipe. This makes it impossible for two arbitrary processes to share the same pipe, unless the pipe was created by a common ancestor process.

This drawback is substantial for many application programs. Consider, for instance, a database engine server, which continuously polls client processes wishing to issue some queries and which sends the results of the database lookups back to them. Each interaction between the server and a given client might be handled by a pipe. However, client processes are usually created on demand by a command shell when a user explicitly queries the database; server and client processes thus cannot easily share a pipe.

To address such limitations, Unix systems introduce a special file type called a named pipe or FIFO (which stands for "first in, first out;" the first byte written into the special file is also the first byte that is read). Each FIFO is much like a pipe: rather than owning disk blocks in the filesystems, an opened FIFO is associated with a kernel buffer that temporarily stores the data exchanged by two or more processes.

Thanks to the disk inode, however, a FIFO can be accessed by every process, because the FIFO filename is included in the system's directory tree. Thus, in our example, the communication between server and clients may be easily established by using FIFOs instead of pipes. The server creates, at startup, a FIFO used by client programs to make their requests. Each client program creates, before establishing the connection, another FIFO to which the server program can write the answer to the query and includes the FIFO's name in the initial request to the server.

In Linux 2.6, FIFOs and pipes are almost identical and use the same pipe_inode_info structures. As a matter of fact, the read and write file operation methods of a FIFO are implemented by the same pipe_read( ) and pipe_write( ) functions described in the earlier sections "Reading from a Pipe" and "Writing into a Pipe." Actually, there are only two significant differences:

-
- FIFO inodes appear on the system directory tree rather than on the pipefs special filesystem.
-
- FIFOs are a bidirectional communication channel; that is, it is possible to open a FIFO in read/write mode.

To complete our description, therefore, we just have to explain how FIFOs are created and opened.

19.2.1. Creating and Opening a FIFO

A process creates a FIFO by issuing a mknod( )[*] system call (see the section "Device Files" in Chapter 13), passing to it as parameters the pathname of the new FIFO and the value S_IFIFO (0x10000) logically ORed with the permission bit mask of the new file. POSIX introduces a function named mkfifo( ) specifically to create a FIFO. This call is implemented in Linux, as in System V Release 4, as a C library function that invokes mknod( ).

[*] In fact, mknod( ) can be used to create nearly every kind of file, such as block and character device files, FIFOs, and even regular files (it cannot create directories or sockets, though).

Once created, a FIFO can be accessed through the usual open( ), read( ), write( ), and close( ) system calls, but the VFS handles it in a special way, because the FIFO inode and file operations are customized and do not depend on the filesystems in which the FIFO is stored.

19.3. System V IPC

IPC is an abbreviation for Interprocess Communication and commonly refers to a set of mechanisms that allow a User Mode process to do the following:

-
- Synchronize itself with other processes by means of semaphores
-
- Send messages to other processes or receive messages from them
-
- Share a memory area with other processes

System V IPC first appeared in a development Unix variant called "Columbus Unix " and later was adopted by AT&T's System III . It is now found in most Unix systems, including Linux.

IPC data structures are created dynamically when a process requests an IPC resource (a semaphore, a message queue, or a shared memory region). An IPC resource is persistent: unless explicitly removed by a process, it is kept in memory and remains available until the system is shut down. An IPC resource may be used by every process, including those that do not share the ancestor that created the resource.

Because a process may require several IPC resources of the same type, each new resource is identified by a 32-bit IPC key, which is similar to the file pathname in the system's directory tree. Each IPC resource also has a 32-bit IPC identifier, which is somewhat similar to the file descriptor associated with an open file. IPC identifiers are assigned to IPC resources by the kernel and are unique within the system, while IPC keys can be freely chosen by programmers.

When two or more processes wish to communicate through an IPC resource, they all refer to the IPC identifier of the resource.

19.3.1. Using an IPC Resource

IPC resources are created by invoking the semget( ), msgget( ), or shmget( ) functions, depending on whether the new resource is a semaphore, a message queue, or a shared memory region.

The main objective of each of these three functions is to derive from the IPC key (passed as the first parameter) the corresponding IPC identifier, which is then used by the process for accessing the resource. If there is no IPC resource already associated with the IPC key, a new resource is created. If everything goes right, the function returns a positive IPC identifier; otherwise, it returns one of the error codes listed in Table 19-7.

Table 19-7. Error codes returned while requesting an IPC identifier

Error code	Description
EACCESS	Process does not have proper access rights
EEXIST	Process tried to create an IPC resource with the same key as one that already exists
EINVAL	Invalid argument in a parameter of semget( ),

19.4. POSIX Message Queues

The POSIX standard (IEEE Std 1003.1-2001) defines an IPC mechanism based on message queues, which is usually known as POSIX message queues . They are much like the System V IPC's message queues already examined in the section "IPC Messages" earlier in this chapter. However, POSIX message queues sport a number of advantages over the older queues:

-
- A much simpler file-based interface to the applications
-
- Native support for message priorities (the priority ultimately determines the position of the message in the queue)
-
- Native support for asynchronous notification of message arrivals, either by means of signals or thread creation
-
- Timeouts for blocking send and receive operations

POSIX message queues are handled by means of a set of library functions, which are shown in Table 19-15.

Table 19-15. Library functions for POSIX message queues

Function names	Description
mq_open( )	Open (optionally creating) a POSIX message queue
mq_close( )	Close a POSIX message queue (without destroying it)
mq_unlink( )	Destroy a POSIX message queue
mq_send( ) , mq_timedsend( )	Send a message to a POSIX message queue; the latter function defines a time limit for the operation
mq_receive( ) , mq_timedreceive( )	Fetch a message from a POSIX message queue; the latter function defines a time limit for the operation
mq_notify( )	Establish an asynchronous notification mechanism for the arrival of messages in an empty POSIX message queue
mq_getattr( ) ,	Respectively get and set attributes of a POSIX message queue (essentially, whether the send and

Chapter 20. Program ExZecution

The concept of a "process," described in Chapter 3, was used in Unix from the beginning to represent the behavior of groups of running programs that compete for system resources. This final chapter focuses on the relationship between program and process. We specifically describe how the kernel sets up the execution context for a process according to the contents of the program file. While it may not seem like a big problem to load a bunch of instructions into memory and point the CPU to them, the kernel has to deal with flexibility in several areas:

Different executable formats

Linux is distinguished by its ability to run binaries that were compiled for other operating systems. In particular, Linux is able to run an executable created for a 32-bit machine on the 64-bit version of the same machine. For instance, an executable created on a Pentium can run on a 64-bit AMD Opteron .

Shared libraries

Many executable files don't contain all the code required to run the program but expect the kernel to load in functions from a library at runtime.

Other information in the execution context

This includes the command-line arguments and environment variables familiar to programmers.

A program is stored on disk as an executable file, which includes both the object code of the functions to be executed and the data on which these functions will act. Many functions of the program are service routines available to all programmers; their object code is included in special files called "libraries." Actually, the code of a library function may either be statically copied into the executable file (static libraries) or linked to the process at runtime (shared libraries, because their code can be shared by several independent processes).

When launching a program, the user may supply two kinds of information that affect the way it is executed: command-line arguments and environment variables. Command-line arguments are typed in by the user following the executable filename at the shell prompt. Environment variables, such as HOME and PATH, are inherited from the shell, but the users may modify the values of such variables before they launch the program.

In the section "Executable Files," we explain what a program execution context is. In the section " Executable Formats," we mention some of the executable formats supported by Linux and show how Linux can change its "personality" to execute programs compiled for other operating systems. Finally, in the section "The exec Functions," we describe the system call that allows a process to start executing a new program.

20.1. Executable Files

Chapter 1 defined a process as an "execution context." By this we mean the collection of information needed to carry on a specific computation; it includes the pages accessed, the open files, the hardware register contents, and so on. An executable file is a regular file that describes how to initialize a new execution context (i.e., how to start a new computation).

Suppose a user wants to list the files in the current directory; he knows that this result can be simply achieved by typing the filename of the */bin/ls*[*] external command at the shell prompt. The command shell forks a new process, which in turn invokes an execve( ) system call (see the section "The exec Functions" later in this chapter), passing as one of its parameters a string that includes the full pathname for the *ls* executable file*/bin/ls*, in this case. The sys_execve( ) service routine finds the corresponding file, checks the executable format, and modifies the execution context of the current process according to the information stored in it. As a result, when the system call terminates, the process starts executing the code stored in the executable file, which performs the directory listing.

[*] The pathnames of executable files are not fixed in Linux; they depend on the distribution used. Several standard naming schemes, such as Filesystem Hierarchy Standard (FHS), have been proposed for all Unix systems.

When a process starts running a new program, its execution context changes drastically because most of the resources obtained during the process's previous computations are discarded. In the preceding example, when the process starts executing */bin/ls*, it replaces the shell's arguments with new ones passed as parameters in the execve( ) system call and acquires a new shell environment (see the later section "Command-Line Arguments and Shell Environment"). All pages inherited from the parent (and shared with the Copy On Write mechanism) are released so that the new computation starts with a fresh User Mode address space; even the privileges of the process could change (see the later section "Process Credentials and Capabilities"). However, the process PID doesn't change, and the new computation inherits from the previous one all open file descriptors that were not closed automatically while executing the execve( ) system call.[*]

[*] By default, a file already opened by a process stays open after issuing an execve( ) system call. However, the file is automatically closed if the process has set the corresponding bit in the close_on_exec field of the files_struct structure (see Table 12-7 in Chapter 12); this is done by means of the fcntl( ) system call.

20.1.1. Process Credentials and Capabilities

Traditionally, Unix systems associate with each process some credentials, which bind the process to a specific user and a specific user group. Credentials are important on multiuser systems because they determine what each process can or cannot do, thus preserving both the integrity of each user's personal data and the stability of the system as a whole.

The use of credentials requires support both in the process data structure and in the resources being protected. One obvious resource is a file. Thus, in the Ext2 filesystem , each file is owned by a specific user and is bound to a group of users. The owner of a file may decide what kind of operations are allowed on that file, distinguishing among herself, the file's user group, and all other users. When a process tries to access a file, the VFS always checks whether the access is legal, according to the permissions established by the file owner and the process credentials .

The process's credentials are stored in several fields of the process descriptor, listed in Table 20-1. These fields contain identifiers of users and user groups in the system, which are usually compared with the corresponding identifiers stored in the inodes of the files being accessed.

20.2. Executable Formats

The standard Linux executable format is named Executable and Linking Format (ELF). It was developed by Unix System Laboratories and is now the most widely used format in the Unix world. Several well-known Unix operating systems, such as System V Release 4 and Sun's Solaris 2, have adopted ELF as their main executable format.

Older Linux versions supported another format named Assembler OUTput Format(a.out); actually, there were several versions of that format floating around the Unix world. It is seldom used now, because ELF is much more practical.

Linux supports many other different formats for executable files; in this way, it can run programs compiled for other operating systems, such as MS-DOS EXE programs or BSD Unix's COFF executables. A few executable formats, such as Java or bash scripts, are platform-independent.

An executable format is described by an object of type linux_binfmt, which essentially provides three methods:

load_binary

Sets up a new execution environment for the current process by reading the information stored in an executable file.

load_shlib

Dynamically binds a shared library to an already running process; it is activated by the uselib( ) system call.

core_dump

Stores the execution context of the current process in a file named core. This file, whose format depends on the type of executable of the program being executed, is usually created when a process receives a signal whose default action is "dump" (see the section "Actions Performed upon Delivering a Signal" in Chapter 11).

All linux_binfmt objects are included in a singly linked list, and the address of the first element is stored in the formats variable. Elements can be inserted and removed in the list by invoking the register_binfmt( ) and unregister_binfmt( ) functions. The register_binfmt( ) function is executed during system startup for each executable format compiled into the kernel. This function is also executed when a module implementing a new executable format is being loaded, while the unregister_binfmt( ) function is invoked when the module is unloaded.

The last element in the formats list is always an object describing the executable format for interpreted scripts . This format defines only the load_binary method. The corresponding load_script( ) function checks whether the executable file starts with the #! pair of characters. If so, it interprets the rest of the first line as the pathname of another executable file and tries to execute it by passing the name of the script file as a parameter.[*]

[*] It is possible to execute a script file even if it doesn't start with the #! characters, as long as the file is written in the language recognized by a command shell. In this case, however, the script is interpreted either by the shell on which the user types the command or by the default Bourne shell *sh*; therefore, the

20.3. Execution Domains

As mentioned in Chapter 1, a neat feature of Linux is its ability to execute files compiled for other operating systems. Of course, this is possible only if the files include machine code for the same computer architecture on which the kernel is running. Two kinds of support are offered for these "foreign" programs:

-
- Emulated execution: necessary to execute programs that include system calls that are not POSIX-compliant
-
- Native execution: valid for programs whose system calls are totally POSIX-compliant

Microsoft MS-DOS and Windows programs are emulated: they cannot be natively executed, because they include APIs that are not recognized by Linux. An emulator such as DOSemu or Wine (which appeared in the example at the end of the previous section) is invoked to translate each API call into an emulating wrapper function call, which in turn uses the existing Linux system calls. Because emulators are mostly implemented as User Mode applications, we don't discuss them further.

On the other hand, POSIX-compliant programs compiled on operating systems other than Linux can be executed without too much trouble, because POSIX operating systems offer similar APIs. (Actually, the APIs should be identical, although this is not always the case.) Minor differences that the kernel must iron out usually refer to how system calls are invoked or how the various signals are numbered. This information is stored in execution domain descriptors of type exec_domain.

A process specifies its execution domain by setting the personality field of its descriptor and storing the address of the corresponding exec_domain data structure in the exec_domain field of the tHRead_info structure. A process can change its personality by issuing a suitable system call named personality( ) ; typical values assumed by the system call's parameter are listed in Table 20-6. Programmers are not expected to directly change the personality of their programs; instead, the personality( ) system call should be issued by the glue code that sets up the execution context of the process (see the next section).

Table 20-6. Personalities supported by the Linux kernel

Personality	Operating system
PER_LINUX	Standard execution domain
PER_LINUX_32BIT	Linux with 32-bit physical addresses in 64-bit architectures
PER_LINUX_FDPIC	Linux program in ELF FDPIC format
PER_SVR4	System V Release 4
PER_SVR3	System V Release 3
PER_SCOSVR3	SCO Unix Version 3.2

20.4. The exec Functions

Unix systems provide a family of functions that replace the execution context of a process with a new context described by an executable file. The names of these functions start with the prefix exec, followed by one or two letters; therefore, a generic function in the family is usually referred to as an exec function.

The exec functions are listed in Table 20-7; they differ in how the parameters are interpreted.

Table 20-7. The exec functions

Function name	PATH search	Command-line arguments	Environment array
execl( )	No	List	No
execlp( )	Yes	List	No
execle( )	No	List	Yes
execv( )	No	Array	No
execvp( )	Yes	Array	No
execve( )	No	Array	Yes

The first parameter of each function denotes the pathname of the file to be executed. The pathname can be absolute or relative to the process's current directory. Moreover, if the name does not include any / characters, the execlp( ) and execvp( ) functions search for the executable file in all directories specified by the PATH environment variable.

Besides the first parameter, the execl( ), execlp( ), and execle( ) functions include a variable number of additional parameters. Each points to a string describing a command-line argument for the new program; as the "l" character in the function names suggests, the parameters are organized in a list terminated by a NULL value. Usually, the first command-line argument duplicates the executable filename. Conversely, the execv( ), execvp( ), and execve( ) functions specify the command-line arguments with a single parameter; as the v character in the function names suggests, the parameter is the address of a vector of pointers to command-line argument strings. The last component of the array must be NULL.

The execle( ) and execve( ) functions receive as their last parameter the address of an array of pointers to environment strings; as usual, the last component of the array must be NULL. The other functions may access the environment for the new program from the external environ global variable, which is defined in the C library.

All exec functions, with the exception of execve( ), are wrapper routines defined in the C library and use execve( ), which is the only system call offered by Linux to deal with program execution.

The sys_execve( ) service routine receives the following parameters:

-
- The address of the executable file pathname (in the User Mode address space)

Appendix A. System Startup

This appendix explains what happens right after users switch on their computersthat is, how a Linux kernel image is copied into memory and executed. In short, we discuss how the kernel, and thus the whole system, is "bootstrapped."

Traditionally, the term bootstrap refers to a person who tries to stand up by pulling his own boots. In operating systems, the term denotes bringing at least a portion of the operating system into main memory and having the processor execute it. It also denotes the initialization of kernel data structures, the creation of some user processes, and the transfer of control to one of them.

Computer bootstrapping is a tedious, long task, because initially, nearly every hardware device, including the RAM, is in a random, unpredictable state. Moreover, the bootstrap process is highly dependent on the computer architecture; as usual in this book, we refer to the 80 x 86 architecture.

A.1. Prehistoric Age: the BIOS

The moment after a computer is powered on, it is practically useless because the RAM chips contain random data and no operating system is running. To begin the boot, a special hardware circuit raises the logical value of the RESET pin of the CPU. After RESET is asserted, some registers of the processor (including cs and eip) are set to fixed values, and the code found at physical address 0xfffffff0 is executed. This address is mapped by the hardware to a certain read-only, persistent memory chip that is often called Read-Only Memory (ROM). The set of programs stored in ROM is traditionally called the Basic Input/Output System (BIOS) in the 80 x 86 architecture, because it includes several interrupt-driven low-level procedures used by all operating systems in the booting phase to handle the hardware devices that make up the computer. Some operating systems, such as Microsoft's MS-DOS , rely on BIOS to implement most system calls.

Once in protected mode (see the section "Segmentation in Hardware" in Chapter 2), Linux does not use BIOS any longer, but it provides its own device driver for every hardware device on the computer. In fact, the BIOS procedures must be executed in real mode, so they cannot share functions even if that would be beneficial.

The BIOS uses Real Mode addresses because they are the only ones available when the computer is turned on. A Real Mode address is composed of a seg segment and an off offset; the corresponding physical address is given by seg*16+off. As a result, no Global Descriptor Table, Local Descriptor Table, or paging table is needed by the CPU addressing circuit to translate a logical address into a physical one. Clearly, the code that initializes the GDT, LDT, and paging tables must run in Real Mode.

Linux is forced to use BIOS in the bootstrapping phase, when it must retrieve the kernel image from disk or from some other external device. The BIOS bootstrap procedure essentially performs the following four operations:

1.

1. Executes a series of tests on the computer hardware to establish which devices are present and whether they are working properly. This phase is often called Power-On Self-Test (POST). During this phase, several messages, such as the BIOS version banner, are displayed.

1. Recent 80 x 86, AMD64, and Itanium computers make use of the Advanced Configuration and Power Interface(ACPI) standard. The bootstrap code in an ACPI-compliant BIOS builds several tables that describe the hardware devices present in the system. These tables have a vendor-independent format and can be read by the operating system kernel to learn how to handle the devices.

2.

2. Initializes the hardware devices. This phase is crucial in modern PCI-based architectures, because it guarantees that all hardware devices operate without conflicts on the IRQ lines and I/O ports. At the end of this phase, a table of installed PCI devices is displayed.

3.

3. Searches for an operating system to boot. Actually, depending on the BIOS setting, the procedure may try to access (in a predefined, customizable order) the first sector (boot sector) of every floppy disk, hard disk, and CD-ROM in the system.

4.

4. As soon as a valid device is found, it copies the contents of its first sector into RAM, starting from physical address 0x00007c00, and then jumps into that address and executes the code just loaded.

A.2. Ancient Age: the Boot Loader

The boot loader is the program invoked by the BIOS to load the image of an operating system kernel into RAM. Let's briefly sketch how boot loaders work in IBM's PC architecture.

To boot from a floppy disk, the instructions stored in its first sector are loaded in RAM and executed; these instructions copy all the remaining sectors containing the kernel image into RAM.

Booting from a hard disk is done differently. The first sector of the hard disk, named the Master Boot Record (MBR), includes the partition table[*] and a small program, which loads the first sector of the partition containing the operating system to be started. Some operating systems, such as Microsoft Windows 98, identify this partition by means of an active flag included in the partition table;[†] following this approach, only the operating system whose kernel image is stored in the active partition can be booted. As we will see later, Linux is more flexible because it replaces the rudimentary program included in the MBR with a sophisticated programthe "boot loader"that allows users to select the operating system to be booted.

[*] Each partition table entry typically includes the starting and ending sectors of a partition and the kind of operating system that handles it.

[†] The active flag may be set through programs such as fdisk.

Kernel images of earlier Linux versionsup to the 2.4 seriesincluded a minimal "boot loader" program in the first 512 bytes; thus, copying a kernel image starting from the first sector made the floppy bootable. On the other hand, kernel images of Linux 2.6 no longer include such boot loader; thus, in order to boot from floppy disk, a suitable boot loader has to be stored in the first disk sector. Nowadays, booting from a floppy is very similar to booting from a hard disk or from a CD-ROM.

A.2.1. Booting Linux from a Disk

A two-stage boot loader is required to boot a Linux kernel from disk. A well-known Linux boot loader on 80 x 86 systems is named LInux LOader (LILO). Other boot loaders for 80 x 86 systems do exist; for instance, the GRand Unified Bootloader (GRUB) is also widely used. GRUB is more advanced than LILO, because it recognizes several disk-based filesystems and is thus capable of reading portions of the boot program from files. Of course, specific boot loader programs exist for all architectures supported by Linux.

LILO may be installed either on the MBR (replacing the small program that loads the boot sector of the active partition) or in the boot sector of every disk partition. In both cases, the final result is the same: when the loader is executed at boot time, the user may choose which operating system to load.

Actually, the LILO boot loader is too large to fit into a single sector, thus it is broken into two parts. The MBR or the partition boot sector includes a small boot loader, which is loaded into RAM starting from address 0x00007c00 by the BIOS. This small program moves itself to the address 0x00096a00, sets up the Real Mode stack (ranging from 0x00098000 to 0x000969ff), loads the second part of the LILO boot loader into RAM starting from address 0x00096c00, and jumps into it.

In turn, this latter program reads a map of bootable operating systems from disk and offers the user a prompt so she can choose one of them. Finally, after the user has chosen the kernel to be loaded (or let a time-out elapse so that LILO chooses a default), the boot loader may either copy the boot sector of the corresponding partition into RAM and execute it or directly copy the kernel image into RAM.

Assuming that a Linux kernel image must be booted, the LILO boot loader, which relies on BIOS routines, performs essentially the following operations:

A.3. Middle Ages: the setup( ) Function

The code of the setup( ) assembly language function has been placed by the linker at offset 0x200 of the kernel image file. The boot loader can therefore easily locate the code and copy it into RAM, starting from physical address 0x00090200.

The setup( ) function must initialize the hardware devices in the computer and set up the environment for the execution of the kernel program. Although the BIOS already initialized most hardware devices, Linux does not rely on it, but reinitializes the devices in its own manner to enhance portability and robustness. setup( ) performs essentially the following operations:

1. In ACPI -compliant systems, it invokes a BIOS routine that builds a table in RAM describing the layout of the system's physical memory (the table can be seen in the boot kernel messages by looking for the "BIOS-e820" label). In older systems, it invokes a BIOS routine that just returns the amount of RAM available in the system.
2. Sets the keyboard repeat delay and rate. (When the user keeps a key pressed past a certain amount of time, the keyboard device sends the corresponding keycode over and over to the CPU.)
3. Initializes the video adapter card.
4. Reinitializes the disk controller and determines the hard disk parameters.
5. Checks for an IBM Micro Channel bus (MCA).
6. Checks for a PS/2 pointing device (bus mouse).
7. Checks for Advanced Power Management (APM) BIOS support.
8. If the BIOS supports the Enhanced Disk Drive Services (EDD), it invokes the proper BIOS procedure to build a table in RAM describing the hard disks available in the system. (The information included in the table can be seen by reading the files in the *firmware/edd* directory of the sysfs special filesystem.)
9. If the kernel image was loaded low in RAM (at physical address 0x00010000), the function moves it to physical address 0x00001000. Conversely, if the kernel image was loaded high in RAM, the function does not move it. This step is necessary because to be able to store the kernel image on a floppy disk and to reduce the booting time, the kernel image stored on disk is compressed, and the decompression routine needs some free space to use as a temporary buffer following the kernel image in RAM.
10. Sets the A20 pin located on the 8042 keyboard controller. The A20 pin is a hack introduced in the 80286 -based systems to make physical addresses compatible with those of the ancient 8088

A.4. Renaissance: the startup_32( ) Functions

There are two different startup_32( ) functions; the one we refer to here is coded in the *arch/i386/boot/compressed/head.S* file. After setup( ) terminates, the function has been moved either to physical address 0x00100000 or to physical address 0x00001000, depending on whether the kernel image was loaded high or low in RAM.

This function performs the following operations:

1. Initializes the segmentation registers and a provisional stack.
2. Clears all bits in the eflags register.
3. Fills the area of uninitialized data of the kernel identified by the _edata and _end symbols with zeros (see the section "Physical Memory Layout" in Chapter 2).
4. Invokes the decompress_kernel( ) function to decompress the kernel image. The "Uncompressing Linux..." message is displayed first. After the kernel image is decompressed, the "O K, booting the kernel." message is shown. If the kernel image was loaded low, the decompressed kernel is placed at physical address 0x00100000. Otherwise, if the kernel image was loaded high, the decompressed kernel is placed in a temporary buffer located after the compressed image. The decompressed image is then moved into its final position, which starts at physical address 0x00100000.
5. Jumps to physical address 0x00100000.

The decompressed kernel image begins with another startup_32( ) function included in the *arch/i386/kernel/head.S* file. Using the same name for both the functions does not create any problems (besides confusing our readers), because both functions are executed by jumping to their initial physical addresses.

The second startup_32( ) function sets up the execution environment for the first Linux process (process 0). The function performs the following operations:

1. Initializes the segmentation registers with their final values.
2. Fills the bss segment of the kernel (see the section "Program Segments and Process Memory Regions" in Chapter 20) with zeros.
3. Initializes the provisional kernel Page Tables contained in swapper_pg_dir and pg0 to identically map the linear addresses to the same physical addresses, as explained in the section "Kernel Page Tables" in Chapter 2.
4. Stores the address of the Page Global Directory in the cr3 register, and enables paging by setting the PG bit in the cr0 register.

A.5. Modern Age: the start_kernel( ) Function

The start_kernel( ) function completes the initialization of the Linux kernel. Nearly every kernel component is initialized by this function; we mention just a few of them:

- The scheduler is initialized by invoking the sched_init( ) function (see Chapter 7).
- The memory zones are initialized by invoking the build_all_zonelists( ) function (see the section "Memory Zones" in Chapter 8).
- The Buddy system allocators are initialized by invoking the page_alloc_init( ) and mem_init( ) functions (see the section "The Buddy System Algorithm" in Chapter 8).
- The final initialization of the IDT is performed by invoking trap_init( ) (see the section "Exception Handling" in Chapter 4) and init_IRQ( ) (see the section "IRQ data structures" in Chapter 4).
- The TASKLET_SOFTIRQ and HI_SOFTIRQ are initialized by invoking the softirq_init( ) function (see the section "Softirqs" in Chapter 4).
- The system date and time are initialized by the time_init( ) function (see the section "The Linux Timekeeping Architecture" in Chapter 6).
- The slab allocator is initialized by the kmem_cache_init( ) function (see the section "General and Specific Caches" in Chapter 8).
- The speed of the CPU clock is determined by invoking the calibrate_delay( ) function (see the section "Delay Functions" in Chapter 6).
- The kernel thread for process 1 is created by invoking the kernel_thread( ) function. In turn, this kernel thread creates the other kernel threads and executes the */sbin/init* program, as described in the section "Kernel Threads" in Chapter 3.

Besides the "Linux version 2.6.11..." message, which is displayed right after the beginning of start_kernel( ), many other messages are displayed in this last phase, both by the init program and by the kernel threads. At the end, the familiar login prompt appears on the console (or in the graphical screen, if the X Window System is launched at startup), telling the user that the Linux kernel is up and running.

B.1. To Be (a Module) or Not to Be?

When system programmers want to add new functionality to the Linux kernel, they are faced with a basic decision: should they write the new code so that it will be compiled as a module, or should they statically link the new code to the kernel?

As a general rule, system programmers tend to implement new code as a module. Because modules can be linked on demand (as we see later), the kernel does not have to be bloated with hundreds of seldom-used programs. Nearly every higher-level component of the Linux kernelfilesystems, device drivers, executable formats, network layers, and so oncan be compiled as a module. Linux distributions use modules extensively in order to support in a seamless way a wide range of hardware devices. For instance, the distribution puts tens of sound card driver modules in a proper directory, although only one of these modules will be effectively loaded on a specific machine.

However, some Linux code must necessarily be linked statically, which means that either the corresponding component is included in the kernel or it is not compiled at all. This happens typically when the component requires a modification to some data structure or function statically linked in the kernel.

For example, suppose the component has to introduce new fields into the process descriptor. Linking a module cannot change an already defined data structure such as task_struct because, even if the module uses its modified version of the data structure, all statically linked code continues to see the old version. Data corruption easily occurs. A partial solution to the problem consists of "statically" adding the new fields to the process descriptor, thus making them available to the kernel component no matter how it has been linked. However, if the kernel component is never used, such extra fields replicated in every process descriptor are a waste of memory. If the new kernel component increases the size of the process descriptor a lot, one would get better system performance by adding the required fields in the data structure only if the component is statically linked to the kernel.

As a second example, consider a kernel component that has to replace statically linked code. It's pretty clear that no such component can be compiled as a module, because the kernel cannot change the machine code already in RAM when linking the module. For instance, it is not possible to link a module that changes the way page frames are allocated, because the Buddy system functions are always statically linked to the kernel.[*]

[*] You might wonder why your favorite kernel component has not been modularized. In most cases, there is no strong technical reason because it is essentially a software license issue. Kernel developers want to make sure that core components will never be replaced by proprietary code released through binary-only "black-box" modules.

The kernel has two key tasks to perform in managing modules. The first task is making sure the rest of the kernel can reach the module's global symbols, such as the entry point to its main function. A module must also know the addresses of symbols in the kernel and in other modules. Thus, references are resolved once and for all when a module is linked. The second task consists of keeping track of the use of modules, so that no module is unloaded while another module or another part of the kernel is using it. A simple reference count keeps track of each module's usage.

B.1.1. Module Licenses

The license of the Linux kernel (GPL, version 2) is liberal in what users and industries can do with the source code; however, it strictly forbids the release of source code derived fromor heavily depending onthe Linux code under a non-GPL license. Essentially, the kernel developers want to be sure that their codeand all the code derived from itwill remain freely usable by all users.

B.2. Module Implementation

Modules are stored in the filesystem as ELF object files and are linked to the kernel by executing the *insmod* program (see the later section, "Linking and Unlinking Modules"). For each module, the kernel allocates a memory area containing the following data:

- A module object
- A null-terminated string that represents the name of the module (all modules must have unique names)
- The code that implements the functions of the module

The module object describes a module; its fields are shown in Table B-1. A doubly linked circular list collects all module objects; the list head is stored in the modules variable, while the pointers to the adjacent elements are stored in the list field of each module object.

Table B-1. The module object

Type	Name	Description
enum module_state	state	The internal state of the module
struct list_head	list	Pointers for the list of modules
char [60]	name	The module name
struct module_kobject	mkobj	Includes a kobject data structure and a pointer to this module object
struct module_param_attrs *	param_attrs	Pointer to an array of module parameter descriptors
const struct kernel_symbol *	syms	Pointer to an array of exported symbols
unsigned int	num_syms	Number of exported symbols
const unsigned long *	crcs	Pointer to an array of CRC values for the exported symbols
const struct kernel_symbol *	gpl_syms	Pointer to an array of GPL-exported symbols

B.3. Linking and Unlinking Modules

A user can link a module into the running kernel by executing the *insmod* external program. This program performs the following operations:

1. Reads from the command line the name of the module to be linked.
2. Locates the file containing the module's object code in the system directory tree. The file is usually placed in some subdirectory below */lib/modules*.
3. Reads from disk the file containing the module's object code.
4. Invokes the init_module( ) system call, passing to it the address of the User Mode buffer containing the module's object code, the length of the object code, and the User Mode memory area containing the parameters of the *insmod* program.
5. Terminates.

The sys_init_module( ) service routine does all the real work; it performs the following main operations:

1. Checks whether the user is allowed to link the module (the current process must have the CAP_SYS_MODULE capability). In every situation where one is adding functionality to a kernel, which has access to all data and processes on the system, security is a paramount concern.
2. Allocates a temporary memory area for the module's object code; then, copies into this memory area the data in the User Mode buffer passed as first parameter of the system call.
3. Checks that the data in the memory area effectively represents a module's ELF object; otherwise, returns an error code.
4. Allocates a memory area for the parameters passed to the *insmod* program, and fills it with the data in the User Mode buffer whose address was passed as third parameter of the system call.
5. Walks the modules list to verify that the module is not already linked. The check is done by comparing the names of the modules (name field in the module objects).
6. Allocates a memory area for the core executable code of the module, and fills it with the contents of the relevant sections of the module.
7. Allocates a memory area for the initialization code of the module, and fills it with the contents of the relevant sections of the module.

B.4. Linking Modules on Demand

A module can be automatically linked when the functionality it provides is requested and automatically removed afterward.

For instance, suppose that the MS-DOS filesystem has not been linked, either statically or dynamically. If a user tries to mount an MS-DOS filesystem, the mount( ) system call normally fails by returning an error code, because MS-DOS is not included in the file_systems list of registered filesystems. However, if support for automatic linking of modules has been specified when configuring the kernel, Linux makes an attempt to link the MS-DOS module, and then scans the list of registered filesystems again. If the module is successfully linked, the mount( ) system call can continue its execution as if the MS-DOS filesystem were present from the beginning.

B.4.1. The modprobe Program

To automatically link a module, the kernel creates a kernel thread to execute the *modprobe* external program,[*] which takes care of possible complications due to module dependencies. The dependencies were discussed earlier: a module may require one or more other modules, and these in turn may require still other modules. For instance, the MS-DOS module requires another module named *fat* containing some code common to all filesystems based on a File Allocation Table (FAT). Thus, if it is not already present, the fat module must also be automatically linked into the running kernel when the MS-DOS module is requested. Resolving dependencies and finding modules is a type of activity that's best done in User Mode, because it requires locating and accessing module object files in the filesystem.

[*] This is one of the few examples in which the kernel relies on an external program.

The *modprobe* external program is similar to *insmod*, since it links in a module specified on the command line. However, *modprobe* also recursively links in all modules used by the module specified on the command line. For instance, if a user invokes *modprobe* to link the MS-DOS module, the program links the fat module, if necessary, followed by the MS-DOS module. Actually, *modprobe* simply checks for module dependencies; the actual linking of each module is done by forking a new process and executing *insmod*.

How does *modprobe* know about module dependencies? Another external program named *depmod* is executed at system startup. It looks at all the modules compiled for the running kernel, which are usually stored inside the */lib/modules* directory. Then it writes all module dependencies to a file named *modules.dep*. The *modprobe* program can thus simply compare the information stored in the file with the list of linked modules yielded by the */proc /modules* file.

B.4.2. The request_module( ) Function

In some cases, the kernel may invoke the request_module( ) function to attempt automatic linking for a module.

Consider again the case of a user trying to mount an MS-DOS filesystem. If the get_fs_type( ) function discovers that the filesystem is not registered, it invokes the request_module( ) function in the hope that MS-DOS has been compiled as a module.

If the request_module( ) function succeeds in linking the requested module, get_fs_type( ) can continue as if the module were always present. Of course, this does not always happen; in our example, the MS-DOS module might not have been compiled at all. In this case, get_fs_type( ) returns an error code.

The request_module( ) function receives the name of the module to be linked as its parameter. It

Bibliography

This bibliography is broken down by subject area and lists some of the most common and, in our opinion, useful books and online documentation on the topic of kernels.

PREV NEXT

Books on Unix Kernels

Bach, M. J. The Design of the Unix Operating System. Prentice Hall International, Inc., 1986. A classic book describing the SVR2 kernel.

Goodheart, B. and J. Cox. The Magic Garden Explained: The Internals of the Unix System V Release 4. Prentice Hall International, Inc., 1994. An excellent book on the SVR4 kernel.

Mauro, J. and R. McDougall. Solaris Internals: Core Kernel Architecture. Prentice Hall, 2000. A good introduction to the Solaris kernel.

McKusick, M. K., M. J. Karels, and K. Bostic. The Design and Implementation of the 4.4 BSD Operating System. Addison Wesley, 1986. Perhaps the most authoritative book on the 4.4 BSD kernel.

Schimmel, Curt. UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers. Addison-Wesley, 1994. An interesting book that deals mostly with the problem of cache implementation in multiprocessor systems.

Vahalia, U. Unix Internals: The New Frontiers. Prentice Hall, Inc., 1996. A valuable book that provides plenty of insight on modern Unix kernel design issues. It includes a rich bibliography.

Books on the Linux Kernel

Beck, M., H. Boehme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, and C. Schroter. Linux Kernel Programming (3rd ed.). Addison Wesley, 2002. A hardware-independent book covering the Linux 2.4 kernel.

Benvenuti, Christian. Understanding Linux Network Internals. O'Reilly Media, Inc., 2006. Covers standard networking protocols and the details of Linux implementation, with a focus on layer 2 and 3 activities.

Corbet, J., A. Rubini, and G. Kroah-Hartman. Linux Device Drivers (3rd ed.). O'Reilly & Associates, Inc., 2005. A valuable book that is somewhat complementary to this one. It gives plenty of information on how to develop drivers for Linux.

Gorman, M. Understanding the Linux Virtual Memory Manager. Prentice Hall PTR, 2004. Focuses on a subset of the chapters included in this book, namely those related to the Virtual Memory Manager.

Herbert, T. F. The Linux TCP/IP Stack: Networking for Embedded Systems (Networking Series). Charles River Media, 2004. Describes in great details the TCP/IP Linux implementation in the 2.6 kernel.

Love, R. Linux Kernel Development (2nd ed.). Novell Press, 2005. A hardware-independent book covering the Linux 2.6 kernel. Some readers suggest to read it before attacking this book.

Mosberger, D., S. Eranian, and B. Perens. IA-64 Linux Kernel: Design and Implementation. Prentice Hall, Inc., 2002. An excellent description of the hardware-dependent Linux kernel for the Itanium IA-64 microprocessor.

Books on PC Architecture and Technical Manuals on Intel Microprocessors

Intel. Intel Architecture Software Developer's Manual, vol. 3: System Programming Guide. 2005. Describes the Intel Pentium microprocessor architecture. It can be downloaded from: http://developer.intel.com/design/processors/pentium4/manuals/25366816.pdf.

Intel. MultiProcessor Specification, Version 1.4. 1997. Describes the Intel multiprocessor architecture specifications. It can be downloaded from http://www.intel.com/design/pentium/datashts/24201606.pdf.

Messmer, H. P. The Indispensable PC Hardware Book (4th ed.). Addison Wesley Professional, 2001. A valuable reference that exhaustively describes the many components of a PC.

Other Online Documentation Sources

Linux source code

The official site for getting kernel source can be found at http://www.kernel.org.Many mirror sites are also available all over the world.

A valuable search engine for the Linux 2.6 source code is available at http://lxr.linux.no.

GCC manuals

All distributions of the GNU C compiler should include full documentation for all its features, stored in several info files that can be read with the Emacs program or an info reader. By the way, the information on Extended Inline Assembly is quite hard to follow, because it does not refer to any specific architecture. Some pertinent information about 80 x 86 GCC's Inline Assembly and *gas*, the GNU assembler invoked by GCC, can be found at:
http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html
http://www.ibm.com/developerworks/linux/library/l-ia.htmlhttp://www.gnu.org/manual/gas-2.9.1/as.html

The Linux Documentation Project

The web site (http://www.tldp.org) contains the home page of the Linux Documentation Project, which, in turn, includes several interesting references to guides, FAQs, and HOWTOs.

Linux kernel development forum

The newsgroup comp.os.linux.development.system is dedicated to discussions about development of the Linux system.

The linux-kernel mailing list

This fascinating mailing list contains much noise as well as a few pertinent comments about the current development version of Linux and about the rationale for including or not including in the kernel some proposals for changes. It is a living laboratory of new ideas that are taking shape. The name of the mailing list is linux-kernel@vger.kernel.org.

The Linux Kernel online book

Authored by David A. Rusling, this 200-page book can be viewed at http://www.tldp.org/LDP/tlk/tlk.html, and describes some fundamental aspects of the Linux 2.0 kernel.

Linux Virtual File System

The page at http://www.safe-mbox.com/~rgooch/linux/docs/vfs.txt is an introduction to the Linux Virtual File System. The author is Richard Gooch.

Research Papers Related to Linux Development

We list here a few papers that we have mentioned in this book. Needless to say, there are many other articles that have a great impact on the development of Linux.

McCreight, E. "Priority Search Tree," SIAM J. Comput., Vol. 14, No 2, pp. 257276, May 1985

Johnson, T. and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," Proceedings of the 20th IEEE VLDB Conf., Santiago, Chile, 1994, pp. 439450.

Bonwick, J. "The Slab Allocator: An Object-Caching Kernel Memory Allocator," Proceedings of Summer 1994 USENIX Conference, pp. 8798.

PREV NEXT

About the Authors

Daniel P. Bovet received his Ph.D. in computer science at UCLA in 1968 and is a full professor at the University of Rome, Tor Vergata, Italy. He had to wait over 25 years before being able to teach an operating systems course in a proper manner, due to the lack of source code for modern, well-designed systems. Now, thanks to cheap PCs and Linux, Dan and Marco are able to cover all the facets of an operating system and can hand out tough, satisfying homework to their students. (These young students working at home on their PCs are really spoiled; they never had to fight with punched cards.) In fact, Dan was so fascinated by the accomplishments of Linus Torvalds and his followers that he spent the last few years trying to unravel some of Linux's mysteries. It seemed natural, after all that work, to write a book about what he found.

Marco Cesati received a degree in mathematics in 1992 and a Ph.D. in computer science at the University of Rome, La Sapienza, in 1995. He is now a research assistant in the computer science department of the School of Engineering at the University of Rome, Tor Vergata. In the past, he has served as a system administrator and Unix programmer for the university (as a Ph.D. student) and for several institutions (as a consultant). During the last few years, he has been continuously involved in teaching his students how to change the Linux kernel in strange and funny ways.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

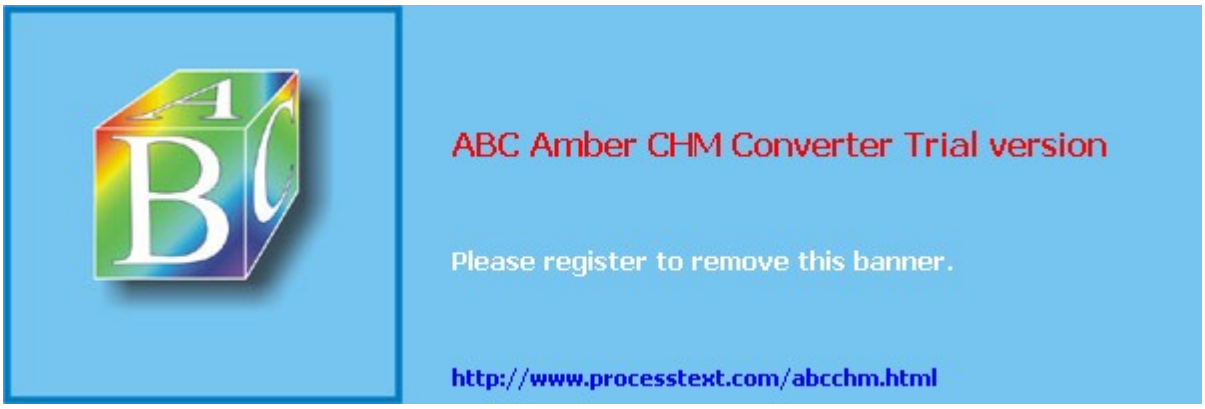
Darren Kelly was the production editor for Understanding the Linux Kernel, Third Edition. Sharon Lundsford was the copyeditor and Julie Campbell was the proofreader. Mary Brady and Claire Cloutier provided quality control. Jansen Fernald and Loranah Dimant provided production assistance. Amy Parker provided production services.

Edie Freedman designed the cover of this book, based on a series design by herself and Hanna Dyer. The cover image of a man with a bubble is a 19th-century engraving from the Dover Pictorial Archive. Karen Montgomery produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. The chapter opening image is from the Dover Pictorial Archive. This book was converted to FrameMaker 5.5.6 by Keith Fahlgren with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano, Jessamyn Read, and Lesley Borash using Macromedia FreeHand 9 and Adobe Photoshop 6.

Index

PREV NEXT



Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

PREV NEXT

Index

PREV NEXT

Index

Index

Index

Index

Index

Index

PREV NEXT

Index

PREV NEXT

Index

HOME

FAQ

UPDATES & NEWS

BUNDLES

FORMATS & CONVERSIONS GUIDE

LINKS

DATABASE

Access Converter
Advantage Converter
Clarion Converter
CSV Converter
DBF Converter
DBISAM Converter
Paradox Converter
OPL Converter

E-MAIL/CHAT/NEWS

Agent Converter
AOL Converter
Barca Converter
Becky Converter
BlackBerry Converter
Calypso Converter
CompuServe Converter
EarthLink Converter
Eudora Converter
GroupWise Converter
iCalendar Converter
ICQ Converter
Incredimail Converter

Buy Now

ABC Amber CHM Converter

Software magazines called ABC Amber CHM Converter "the best ever tool to convert your chm documentation into pdf format".

Most likely, **ABC Amber CHM Converter** is a batch decompiler for Compiled Windows HTML Help files (*.chm) you've been searching for a long time. Taking CHM files or CHM ebooks, it will convert them to any document format you wish - PDF, RTF, HTML, DOC, HLP, TXT, DBF, XML, CSV, XLS, MDB, etc.

It creates the contents with bookmarks (in PDF, RTF, DOC and HTML), extracts pictures, keeps hyperlinks and the structure of CHM file. In other words, you can produce a new hypertext system in most popular formats.

Key features of conversion to PDF using ABC Amber CHM Converter program include 40/128 bits PDF encryption, advanced PDF security options, page size and page orientation support, resolution mode, compression mode, and more.

Also you can use this program as a **chm2web** and **chm2hlp** tool. As all "ABC Amber" products, it's really easy as ABC and powerful as Amber. Currently our software supports more than 50 languages.

No doubt this powerful tool helps you save your time and reduce your efforts.

Partial Features List

- reads **CHM** files and converts them to **PDF** (doesn't require Adobe Acrobat to be installed), **HTML** (single file and web-site), **RTF** (MS Word doesn't need to be installed), **HLP**, **TXT** (ANSI and Unicode), **DOC** (MS Word), **DBF**, **MDB** (MS Access), **CSV**, **XML**, **XLS** (MS Excel), Clipboard
- supports a batch conversion (registered version only)
- generates contents with bookmarks and hyperlinks in the output file
- supports multiple charsets and encoding tables
- easy to use and easy to set up
- extracts pictures, maintains hyperlinks
- keeps the structure of CHM file
- processes bookmarks, frames and scripts in CHM file
- supports multiple PDF and HLP export options
- exports to **HJT** (TreePad) and **KNT** (KeyNote) (since 4.08)

Juno Converter
Lotus Notes Converter
Mozilla (Netscape) Converter
Opera Converter
Outlook Converter
Outlook Express Converter
DBX Converter (MS OE)
Pegasus Converter
PocoMail Converter
SeaMonkey Converter
SunBird Converter
The Bat! Converter
TBB Converter (The Bat!)
Thunderbird Converter
T-Online Converter
Windows Mail Converter
Windows Live Mail Converter

IMAGE/TEXT
AutoCad Converter
DICOM Converter
Image Converter
Image2Text Converter
Kodak Converter
Paintshop Converter
PDF2Image Converter
Photoshop Converter
SVG Converter
Text2Image Converter

HELP FILES
CHM Converter
CHM Merger
HLF Converter
HLP Converter
HxS Converter
ScrapBook Converter

OFFICE
HTML2Excel Converter
OneNote Converter
PowerPoint Converter

- supports **HTB** files (since 5.09)
- creates annotations in PDF for java pop-ups (since 6.34)
- **bonus:** converts CHM to LIT (MS Reader), RB (Rocket eBook), FB2 (FictionBook) and PDB (Palm) (since 6.01), to **IPD** (BlackBerry) (since 6.23), to **MHT** (single Web Archive) (since 6.28)
- multi-language support, command line support, skin support and more!

Since 2.18 the program can convert all topics, convert starting from the selected topic or convert only selected topic.
Since 3.03 it can process merged CHMs.

Tips

There is the option to **increase the speed of conversion.**
Set "HTML import" option to embedded parser (Tools - Options - General), open your chm file and then convert it. However this is not the recommended way (external importer gives the better results - supports CSS styles, etc.)

Q: I am converting my CHM e-books into PDF for my iRex Iliad. I need to have a custom PDF page size (124mm x 152mm) in order to have the best output. Is there any way that I can get Amber to do a custom PDF size?
A: Select Tools - Options - PDF, select "custom size" option and set PDF page width and height.

Other tips

Screen Shot

Project Converter
Publisher Converter
Visio Converter
Word2Excel Converter

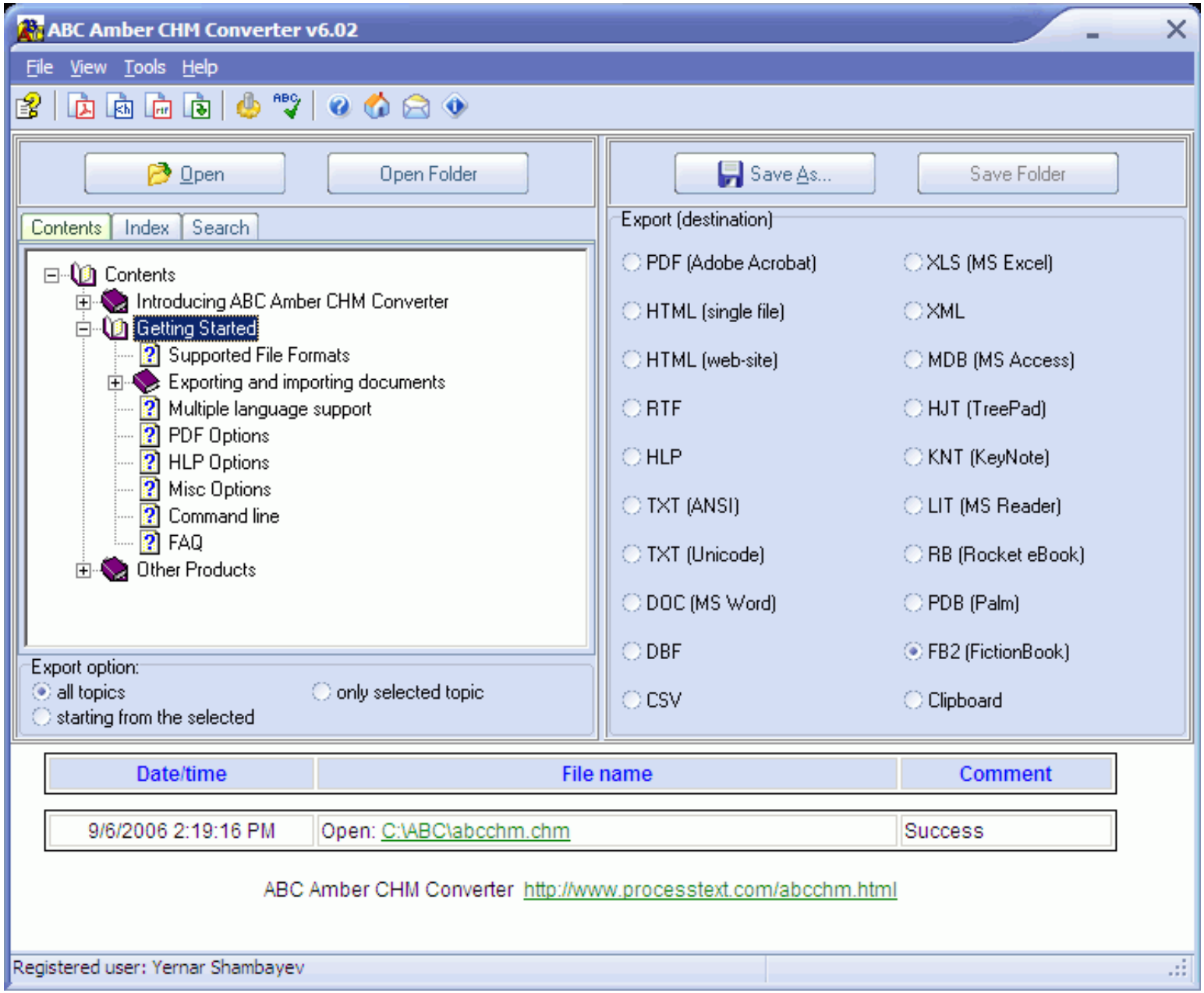
TEXT
Gemstar Converter
Palm Converter
PDF Converter
PDF Merger
Rocket eBook Converter
Sony Converter
Text Converter
Text Merger
Text2Mail Converter
TEX Converter
WordPerfect Converter

SPREADSHEET
Excel Converter
QuattroPro Converter
Lotus 1-2-3 Converter

OTHER
IPD Merger
KeyNote Converter
Nokia Converter
Projekt Converter
SPSS Converter
TreePad Converter
XML Converter

DEVELOPER
Cobol Converter
Pascal Converter

SERVER
DB/2 Converter
Firebird Converter
Interbase Converter
MS SQL Converter



Download Demo Version

To download free 30-day demo version, click here (if you are a registered user, please don't use this link. Use only the link for registered users - see a registration email).

With the demo version of the program, you may test it free of charge for 30 days. If you want to continue to use it after the 30 days,

MySQL Converter
Oracle Converter
PostgreSQL Converter
Sybase Converter

GAME
PGN (Chess) Converter

FREE
Absolute Converter
Audio Converter
BlackBerry Editor
CD Converter
DBA Converter
DJVU Converter
EPS Converter
Eureka Email Converter
Evolution Converter
Excel2BlackBerry Converter
Flash Converter
Gmail Converter
Google PageRank Converter
ICL Converter
IE Converter
IP Converter
Karaoke Converter
KomaMail Converter
Living CookBook Converter
LIT Converter
Ltrack Converter
MasterCook Converter
Measure Converter
Mulberry Converter
Psion Converter
SiMail Converter
Soccer Converter
SQLite Converter
Sylpheed Converter
Torrent Converter
vCard Converter
Winmail Converter
Zip Batch Extractor

you must register the program. Note that the demo version doe not support batch conversion, etc. After purchasing the program we'll send you the fully functional and **LATEST** version (v7.12) with no restrictions.

How To Order

You can order ABC Amber CHM Converter over the Internet from RegNow, SWREG or Emetrix. The ordering page is on a secure server, ensuring that your confidential information remains confidential. If you have any problems, please don't hesitate to contact us.

License for Full Version

SINGLE
License for one user.
Price: US $19.95
Order page: SWREG or RegNow or Emetrix

GROUP
License for 2 or 3 users.
Price: US $29.95
Order page: SWREG

SITE
License for unlimited number of users under the same company.
Price: US $59.95
Order page: SWREG

What our customers are saying:
"I just purchased CHM Converter. The program works wonderfully. I need printed documentation for some of the programs I purchase. This program solves the problem for me."
"Nice, simple and inexpensive products that just do what they are supposed to do."
"This is one of the very useful software I have seen for a while."
"I like your software. I'm very impressed by the speed with which it processes the chm to html (site) conversions."
"This program is fantastic! Thanks!"
"Thank you for the great products and an exceptional support response!"
"This software will come really handy."
"Gratulations for this great tool."
"Thanks to your software which makes it a whole lot easier than some of the other programs that are out there!"
"Best support I've ever received from any software company!"

"Absolutely dynamite trial demo. Must have this tool."
"Have installed and used the app. It's wonderful. Nice piece of coding."
"Tried your trial version and liked it. Thank you for your products, they are good and reasonably priced."
"Thanks, we were very impressed with the demo version - it will save us a lot of trouble."

"The XML conversion is great! I'm currently beta testing and reviewing the CHM files. Getting a new one every week or so. With your tool and a decent compare utility, I was able to quickly see what content changes were made to update my corresponding Word document way easier than comparing word documents directly."

Review

"Whether we like to admit it or not, we all need help sometimes. This is certainly true with all of the different types of software floating around out there today. The reality is that most of us are not really that fond of the typical .chm files that hold the "secrets" to that software we are trying to decipher. Using a tool like the ABC Amber CHM Converter might just be the ticket however. Simple to use, it allows you to convert .chm files into Word, Text and a variety of other formats that may be more to your liking." (Lockergnome)

home | back to top