Freely ye have received, freely give.

—Matthew 10:8—

The world is moving so fast these days that the man who says it can't be done is generally interrupted by someone doing it.

—Elbert Hubbard—

When your Daemon is in charge, do not try to think consciously. Drift, wait and obey.

—Rudyard Kipling—

I long to accomplish a great and noble task, but it is my chief duty to accomplish small tasks as if they were great and noble.

—Helen Keller—

Our children may learn about heroes of the past. Our task is to make ourselves architects of the future.

—Jomo Mzee Kenyatta—

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Chapter 20

Case Study: Linux

Objectives

After reading this chapter, you will understand:

- *Linux kernel architecture.*
- *the Linux implementation of operating system components such as process, memory and file management.*
- *the software layers that compose the Linux kernel.*
- *how Linux organizes and manages system devices.*
- *how Linux manages I/O operations.*
- *interprocess communication and synchronization mechanisms in Linux.*
- *how Linux scales to multiprocessor and embedded systems.*
- *Linux security features.*

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Chapter Outline

20.1
Introduction

20.2
History

20.3
Linux Overview

20.3.1 Development and Community

20.3.2 Distributions

20.3.3 User Interface

20.3.4 Standards

20.4
Kernel Architecture

20.4.1 Hardware Platforms

Mini Case Study: User-Mode Linux (UML)

20.4.2 Loadable Kernel Modules

20.5
Process Management

20.5.1 Process and Thread Organization

20.5.2 Process Scheduling

20.6
Memory Management

20.6.1 Memory Organization

20.6.2 Physical Memory Allocation and Deallocation

20.6.3 Page Replacement

20.6.4 Swapping

20.7
File Systems

20.7.1 Virtual File System

20.7.2 Virtual File System Caches

20.7.3 Second Extended File System (ext2fs)

20.7.4 Proc File System

20.8
Input/Output Management

20.8.1 Device Drivers

20.8.2 Character Device I/O

20.8.3 Block Device I/O

20.8.4 Network Device I/O

20.8.5 Unified Device Model

20.8.6 Interrupts

20.9
Kernel Synchronization

20.9.1 Spin Locks

20.9.2 Reader/Writer Locks

20.9.3 Seqlocks

20.9.4 Kernel Semaphores

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.10 Interprocess Communication
20.10.1 Signals
20.10.2 Pipes
20.10.3 Sockets
20.10.4 Message Queues
20.10.5 Shared Memory
20.10.6 System V Semaphores

20.11 Networking
20.11.1 Packet Processing
20.11.2 Netfilter Framework and Hooks

20.12 Scalability
20.12.1 Symmetric Multiprocessing (SMP)
20.12.2 Nonuniform Memory Access (NUMA)
20.12.3 Other Scalability Features
20.12.4 Embedded Linux

20.13 Security
20.13.1 Authentication
20.13.2 Access Control Methods
20.13.3 Cryptography

Web Resources | Key Terms | Exercises | Recommended Reading | Works Cited

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.1 Introduction

The Linux kernel version 2.6 is the core of the most popular open-source, freely distributed, full-featured operating system. Unlike that of proprietary operating systems, Linux source code is available to the public for examination and modification and is free to download and install. As a result, users of the operating system benefit from a community of developers actively debugging and improving the kernel, an absence of licensing fees and restrictions, and the ability to completely customize the operating system to meet specific needs. Though Linux is not centrally produced by a corporation, Linux users can receive technical support for a fee from Linux vendors or for free through a community of users.

The Linux operating system, which is developed by a loosely organized team of volunteers, is popular in high-end servers, desktop computers and embedded systems. Besides providing core operating system features, such as process scheduling, memory management, device management and file system management, Linux supports many advanced features such as symmetric multiprocessing (SMP), nonuniform memory access (NUMA), access to multiple file systems and support for a broad spectrum of hardware architectures. This case study offers the reader an opportunity to evaluate a real operating system in substantial detail in the context of the operating system concepts discussed throughout this book.

20.2 History

In 1991, Linus Torvalds, a 21-year-old student at the University of Helsinki, Finland, began developing the Linux (the name is derived from "Linus" and "UNIX") kernel as a hobby. Torvalds wished to improve upon the design of Minix, an educational operating system created by Professor Andrew S. Tanenbaum of the Vrije Universiteit in Amsterdam. The Minix source code, which served as a starting point for Torvalds's Linux project, was publicly available for professors to demonstrate basic operating system implementation concepts to their students.[1]

In the early stages of development, Torvalds sought advice about the shortcomings of Minix from those familiar with it. He designed Linux based on these suggestions and made further efforts to involve the operating systems community in his project. In September of 1991, Torvalds released the first version (0.01) of the Linux operating system, announcing the availability of his source code to a Minix newsgroup.[2]

The response led to the creation of a community that has continued to develop and support Linux. Developers downloaded, tested, and modified the Linux code, submitting bug fixes and feedback to Torvalds, who reviewed them and applied the improvements to the code. In October, 1991, Torvalds released version 0.02 of the Linux operating system.[3]

Although early Linux kernels lacked many features implemented in well-established operating systems such as UNIX, developers continued to support the concept of a new, freely available operating system. As Linux's popularity grew,

©1981–2004, Deitel & Associates, Inc. All rights reserved.

developers worked to remedy its shortcomings, such as the absence of a login mechanism and its dependence on Minix to compile. Other missing features were floppy disk support and a virtual memory system.[4] Torvalds continued to maintain the Linux source code, applying changes as he saw fit.

As Linux evolved and drew more support from developers, Torvalds recognized its potential to become more than a hobby operating system. He decided that Linux should conform to the POSIX specification to enhance its interoperability with other UNIX-like systems. Recall that POSIX, the Portable Operating System Interface, defines standards for application interfaces to operating system services, as discussed in Section 2.7, Application Programming Interfaces (API).[5]

The 1994 release of Linux version 1.0 included many features commonly found in a mature operating system, such as multiprogramming, virtual memory, demand loading and TCP/IP networking.[6] It provided the functionality necessary for Linux to become a viable alternative to the licensed UNIX operating system.

Though it benefited from free licensing, Linux suffered from a complex installation and configuration process. To allow users unfamiliar with the details of Linux to conveniently install and use the operating system, academic institutions, such as the University of Manchester and Texas A&M University, and organizations such as Slackware Linux (`www.slackware.org`), created Linux **distributions**, which included software such as the Linux kernel, system applications (e.g., user account management, network management and security tools), user applications (e.g., GUIs, Web browsers, text editors, e-mail applications, databases, and games) and tools to simplify the installation process.[7]

As kernel development progressed, the project adopted a version numbering scheme. The first digit is the **major version number**, which is incremented at Torvalds's discretion for each kernel release that contains a feature set significantly different from that of the previous version. Kernels that are described by an even **minor version number** (the digit directly following the first decimal point), such as version 1.0.9, are considered to be stable releases, whereas an odd minor version number, such as 2.1.6, indicates a development version. The digit following the second decimal point is incremented for each minor update to the kernel.

Development kernels include new features that have not been extensively tested, so they are not sufficiently reliable for production use. Throughout the development process, developers create and test new features; then, once a development kernel becomes stable (i.e., the kernel does not contain any known bugs), Torvalds declares it a release kernel.

By the 1996 release of version 2.0, the Linux kernel had grown to over 400,000 lines of code.[1] Thousands of developers had contributed features and bug fixes, and more than 1.5 million users had installed the operating system.[9] Although this release was appealing to the server market, the vast majority of desktop users were

1 Red Hat version 6.2, which included version 2.0 of the Linux kernel, contained approximately 17 million lines of code. By comparison, Microsoft Windows 95 contained approximately 15 million lines of code and Sun Solaris approximately 8 million. [8]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

still reluctant to use Linux as a client operating system. Version 2.0 provided enterprise features such as support for SMP, network traffic control and disk quotas. Another important feature allowed portions of the kernel to be modularized, so that users could add device drivers and other system components without rebuilding the kernel.

Version 2.2 of the kernel, which was released by Torvalds in 1999, improved the performance of existing 2.0 features, such as SMP, audio support and file systems, and added new features such as an extension to the kernel's networking subsystem that allowed system administrators to inspect and control network traffic at the packet level. This feature simplified firewall installation and network traffic forwarding, as requested by server administrators.[10]

Many new features in version 2.2, such as USB support, CD-RW support and advanced power management, targeted the desktop market. These features were labeled as experimental, because they were not sufficiently reliable for use in production systems. Although version 2.2 improved usability in desktop environments, Linux could not yet truly compete with popular desktop operating systems of the time, such as Microsoft's Windows 98. The desktop user was more concerned with the availability of applications and the "look and feel" of the user interface than with kernel functionality. However, as Linux kernel development continued, so did the development of Linux applications.

The next stable kernel, version 2.4, was released by Torvalds in January, 2001. In this release a number of kernel subsystems were modified and, in some cases, completely rewritten to support newer hardware and to use existing hardware more efficiently. In addition, Linux was modified to run on high-performance architectures including Intel's 64-bit Itanium, 64-bit MIPS and AMD's 64-bit Opteron, and handheld-device architectures such as SuperH.

Enterprise systems companies such as IBM and Oracle had become increasingly interested in Linux as it continued to stabilize and spread to new platforms. Viability in the enterprise systems market, however, required Linux to scale to both high-end and embedded systems, a need fulfilled by version 2.4.[11]

Version 2.4 addressed a critical scalability issue by improving performance on high-end multiprocessor systems. Although Linux had included SMP support since version 2.0, inefficient synchronization mechanisms and other issues limited performance on systems containing more than four processors. Improvements in version 2.4 enabled the kernel to scale to 8, 16 or more processors.[12]

Version 2.4 also addressed the needs of desktop users. Experimental features in the 2.2 kernel, such as USB support and power management, matured in the 2.4 kernel. This kernel supported a large set of desktop devices; however, a variety of issues, such as Microsoft's market power and the small number of user-friendly Linux applications, prevented widespread Linux use on desktop computers.

Development of the version 2.6 kernel focused on scalability, standards compliance and modifications to kernel subsystems to improve performance. Kernel developers focused on scalability by increasing SMP support, providing support for

©1981–2004, Deitel & Associates, Inc. All rights reserved.

NUMA systems and rewriting the process scheduler to increase the performance of scheduling operations. Other kernel enhancements included support for advanced disk scheduling algorithms, a new block I/O layer, improved POSIX compliance, an updated audio subsystem and support for large memories and disks.

20.3 Linux Overview

Linux has a distinct development process and benefits from a wealth of diverse (and free) system and user applications. In this section we summarize Linux kernel features, discuss the process of standardizing and developing the kernel, and introduce several user applications that improve Linux usability and productivity.

In addition to the kernel, Linux systems include user interfaces and applications. A user interface can be as simple as a text-based shell, though standard Linux distributions include a number of GUIs through which users can interact with the system. The Linux operating system borrows from the UNIX layered system approach. Users access applications via a user interface; these applications access resources via a system call interface, thereby invoking the kernel. The kernel may then access the system's hardware, as appropriate, on behalf of the requesting application. In addition to creating user processes, the system creates kernel threads that perform many kernel services. Kernel threads are implemented as **daemons**, which remain dormant until the scheduler or another component of the kernel wakes them.

Because Linux is a multiuser system, the kernel must provide mechanisms to manage user access rights and provide protection for system resources. Therefore, Linux restricts operations that may damage the kernel and/or the system's hardware to a user that has **superuser** (also called **root**) privileges. For example, the superuser privilege enables a user to manage passwords, specify access rights for other users and execute code that modifies system files.

20.3.1 Development and Community

The Linux project is maintained by Linus Torvalds, who is the final arbiter of any code submitted for the kernel. The community of developers constantly modifies the operating system and every two or three years releases a new stable version of the kernel. The community then shifts to the development of the next kernel, discussing new features via e-mail lists and online forums. Torvalds delegates maintenance of stable kernels to trusted developers and manages the development kernel. Bug fixes and performance enhancements for stable releases are applied to the source code and released as updates to the stable version. In parallel, development kernels are released at various stages of the coding process for public review, testing and feedback.

Torvalds and a team of approximately 20 members of his "inner circle"—a set of developers who have proven their competency by producing significant additions to the Linux kernel—are entrusted with enhancing current features and coding new

©1981–2004, Deitel & Associates, Inc. All rights reserved.

ones. These primary developers submit code to Torvalds, who reviews and accepts or rejects it, depending on such factors as correctness, performance and style. When a development kernel has matured to a point at which Torvalds is satisfied with the content of its feature set, he will declare a **feature freeze**. Developers may continue to submit bug fixes, code that improves system performance and enhancements to features that are under development.[13] When the kernel is near completion, a **code freeze** occurs. During this phase only code that fixes bugs is accepted. When Torvalds decides that all important known bugs have been addressed, the kernel is declared stable and is released with a new, even kernel minor version number.

Though many Linux developers contribute to the kernel as individuals, corporations such as IBM have invested significant resources in improving the Linux kernel for use in large-scale systems. Such corporations typically charge for tools and support services. Free support is provided by other users and developers in the Linux community. Users may ask questions in user groups, electronic mailing lists (also called listservs) or forums, and may find answers to questions in FAQs (frequently asked questions) and HOWTOs (step-by-step guides). URLs for such resources can be found via the sites in the Web Resources section at the end of the chapter. Alternatively, dedicated support services can be purchased from vendors.

Linux is free for users to download, modify and distribute under the GNU General Public License (GPL). GNU (pronounced *guh-knew*) is a project created by the Free Software Foundation in 1984 that aims to provide free UNIX-like operating systems and software to the public.[14] The General Public License specifies that any distribution of the software under its license must be accompanied by the GPL, must clearly indicate that the original code has been modified and must include the complete source code. Although Linux is free software, it is copyrighted (many of the copyrights are held by Linus Torvalds); any software that borrows from Linux's copyrighted material must clearly credit its source and must also be distributed under the terms of the GPL.

20.3.2 Distributions

By the end of the 1990s, Linux had matured but was still largely ignored by desktop users. In a PC market dominated by Microsoft and Apple, Linux was considered too difficult to use. Those who wished to install Linux were required to download the source code, manually customize configuration files and compile the kernel. Users still needed to download and install applications to perform productive work. As Linux matured, developers realized a need for a friendly installation process, which led to the creation of distributions that included the kernel, applications and user interfaces as well as other tools and accessories.

Currently more than 300 distributions are available, each providing a variety of features. User-friendly and application-rich distributions are popular among users—they often include an intuitive GUI and productivity applications such as word processors, spreadsheets and Web browsers. Distributions are commonly divided into **packages**, each containing a single application or service. Users can customize a

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Linux system by installing or removing packages, either during the installation process or at runtime. Examples of such distributions are Debian, Mandrake, Red Hat, Slackware and SuSE.[15] Mandrake, Red Hat and SuSE are commercial organizations that provide Linux distributions for markets such as high-end servers and desktop users.[16, 17, 18] Debian and Slackware are nonprofit organizations comprised of volunteer developers who update and maintain Linux distributions.[19, 20] Other distributions tailor to specific environments, such as handheld systems (e.g., OpenZaurus) and embedded systems (e.g., uClinux).[21, 22] All parts of distributions using GPL-licensed code can be freely modified and redistributed by end-users, but the GPL does not prohibit distributors from charging a fee for distribution costs (e.g., the cost of packaging materials) or technical support.[23, 24]

20.3.3 User Interface

In a Microsoft Windows XP or Macintosh OS X environment, the user is presented with a standard, customizable user interface composed of the GUI and an emulated terminal or shell (e.g., a window containing a command-line prompt). On the contrary, Linux is simply the kernel of an operating system and does not specify a "standard" user interface. Many console shells, such as *bash* (Bourne-again shell), *csh* (a shell providing C-like syntax, pronounced "seashell") and *esh* (easy shell) are commonly found on user systems.[25]

For users who prefer a graphical interface to console shells, there are several freely available GUIs, many of which are packaged as part of most Linux distributions. Those most commonly found in Linux systems are composed of several layers. In most Linux systems, the lowest layer is the X Window System (`www.XFree86.org`), a low-level graphical interface originally developed at MIT in 1984.[26] The X Window System provides to higher layers the mechanisms necessary to create and manipulate windows and other graphical components. The second layer of the GUI is the **window manager**, which builds on mechanisms in the X Window System interface to control the placement, appearance, size and other attributes of windows. An optional third layer is called the **desktop environment**. The most popular desktop environments are KDE (K Desktop Environment) and GNOME (GNU Network Object Model Environment). Desktop environments tend to provide a file management interface, tools to facilitate access to common applications and utilities, and a suite of software, typically including Web browsers, text editors and e-mail applications.[27]

20.3.4 Standards

A more recent goal of the Linux operating system has been to conform to a variety of widely recognized standards to improve compatibility between applications written for UNIX-like operating systems and Linux. The most prominent set of standards to which Linux developers strive to conform is POSIX (`standards.ieee.org/regauth/posix/`). Two other sets prominent in UNIX-like operating systems are the Single UNIX Specification (SUS) and the Linux Standards Base (LSB).

©1981–2004, Deitel & Associates, Inc. All rights reserved.

The **Single UNIX Specification** (www.unix.org/version3/) is a suite of standards that define user and application programming interfaces for UNIX operating systems, shells and utilities. Version 3 of the SUS combines several standards (including POSIX, ISO standards and previous versions of the SUS) into one.[28] The Open Group (www.unix.org), which holds the trademark rights and defines standards for the UNIX brand, maintains SUS. To bear the UNIX trademarked name, an operating system must conform to the SUS; The Open Group certifies SUS conformance for a fee.[29]

The **Linux Standard Base** (www.linuxbase.org) is a project that aims to standardize Linux so that applications written for one LSB-compliant distribution will compile and behave exactly the same on any other LSB-compliant distribution. The LSB maintains general standards that apply to elements of the operating system, including libraries, package format and installation, commands and utilities. For example, the LSB specifies a standard file system structure. The LSB also maintains architecture-specific standards that are required for LSB certification. Those who wish to test and certify a distribution for LSB compliance can obtain the tools and certification from the LSB organization for a fee.[30]

Until recently, standards compliance has been a low priority for the kernel, because most kernel developers are concerned with improving the feature set and reliability of Linux. Consequently, most kernel releases do not conform to any one set of standards. During the development of the version 2.6 Linux kernel, developers modified several interfaces to improve compliance with the POSIX, SUS and LSB standards.

20.4 Kernel Architecture

Although Linux is a monolithic kernel (see Section 6.13, Operating System Architectures), recent scalability enhancements have included modular capabilities similar to those supported by microkernel operating systems.[31] Linux is commonly referred to as a UNIX-like or a UNIX-based operating system because it provides many services that characterize UNIX systems, such as AT&T's UNIX System V and Berkeley's BSD. Linux is composed of six primary subsystems: process management, interprocess communication, memory management, file system management, I/O management and networking. These six subsystems are responsible for controlling access to system resources (Fig. 20.1). In the following sections, we examine these kernel subsystems and their interactions.

Process execution on a Linux system occurs in either user mode or kernel mode. User processes run in user mode and must therefore access kernel services via the system call interface. When a user process issues a valid system call (in user mode), the kernel executes the system call in kernel mode on behalf of the process. If the request is invalid (e.g., a process attempts to write to a file that is not open), the kernel returns an error.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

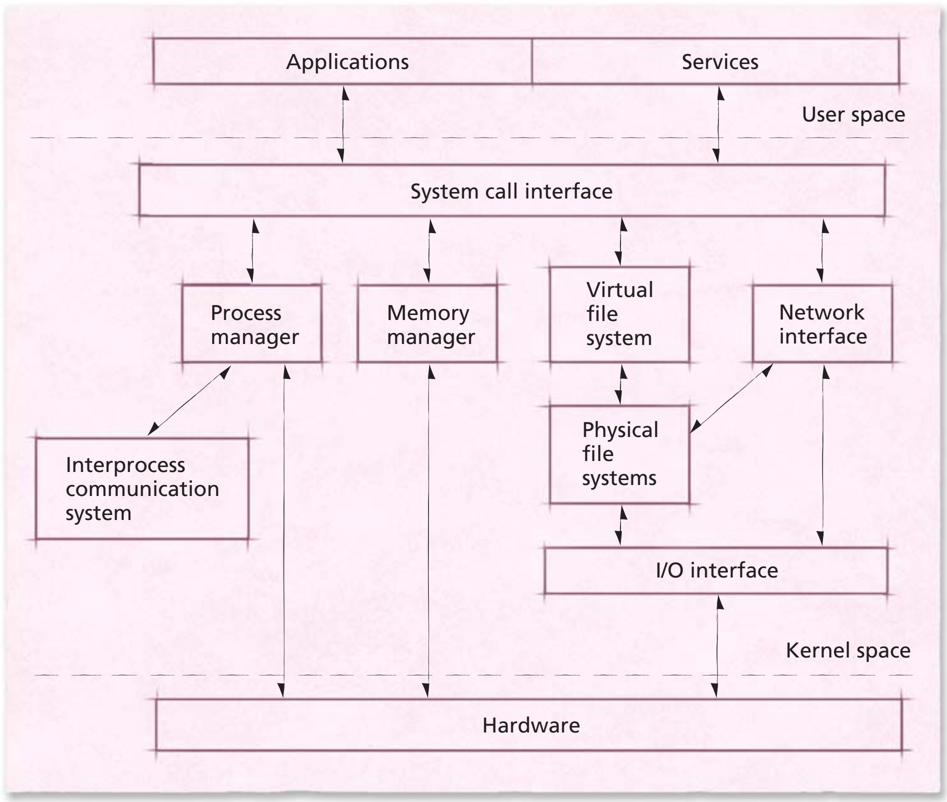


Figure 20.1 | *Linux architecture.*

The process manager is a fundamental Linux subsystem that is responsible for creating processes, providing access to the system's processor(s) and removing processes from the system upon completion (see Section 20.5, Process Management). The kernel's interprocess communication (IPC) subsystem allows processes to communicate with one another. This subsystem interacts with the process manager to permit information sharing and message passing using a variety of mechanisms, discussed in Section 20.10, Interprocess Communication.

The memory management subsystem provides processes with access to memory. Linux assigns each process a virtual memory address space, which is divided into the user address space and the kernel address space. Including the kernel address space within each execution context reduces the cost of context switching from user mode to kernel mode because the kernel can access its data from every user process's virtual address space. The algorithms to manage free (i.e., available) memory and select pages for replacement are discussed in Section 20.6, Memory Management.

Users access files and directories by navigating the directory tree. The root of the directory tree is called the root directory. From the root directory, users can navigate any available file systems. User processes access file system data through

©1981–2004, Deitel & Associates, Inc. All rights reserved.

the system call interface. When system calls access a file or directory in the directory tree, they do so through the **virtual file system (VFS)** interface, which provides to processes a single interface to access files and directories stored in multiple heterogeneous file systems (e.g., ext2 and NFS). The virtual file system passes requests to particular file systems, which manage the layout and location of data, as discussed in Section 20.7, File Systems.

Based on the UNIX model, Linux treats most devices as files, meaning that they are accessed using the same mechanisms with which data files are accessed. When user processes read from or write to devices, the kernel passes requests to the virtual file system interface, which then passes requests to the I/O interface. The I/O interface passes requests to device drivers that perform I/O operations on the hardware in a system. In Section 20.8, Input/Output Management, we discuss the I/O interface and its interaction with other kernel subsystems.

Linux provides a networking subsystem to allow processes to exchange data with other networked computers. The networking subsystem accesses the I/O interface to send and receive packets using the system's networking hardware. It allows applications and the kernel to inspect and modify packets as they traverse the system's networking layers via the packet filtering interface. This interface allows systems to implement firewalls, routers and other network utilities. In Section 20.11, Networking, we discuss the various components of the networking subsystem and their implementations.

20.4.1 Hardware Platforms

Initially, Torvalds developed Linux for use on 32-bit Intel x86 platforms. As its popularity grew, developers implemented Linux on a variety of other architectures. The Linux kernel supports the following platforms: x86 (including Intel IA-32), HP/Compaq Alpha AXP, Sun SPARC, Sun UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, IBM S/390 and zSeries, MIPS, HP PA-RISC, Intel IA-64, AMD x86-64, H8/300, V850 and CRIS.[32]

Each architecture typically requires that the kernel use a different set of low-level instructions to perform operating system functions. For example, an Intel processor implements a different system call mechanism than a Motorola processor. The code that performs operations that are implemented differently across architectures is called **architecture-specific code**. The process of modifying the kernel to support new architecture is called **porting**. To facilitate the process of porting Linux to new platforms, architecture-specific code is separated from the rest of the kernel code into the `/arch` directory of the kernel source tree. The kernel **source tree** organizes each significant component of the kernel into different subdirectories. Each subdirectory in `/arch` contains code corresponding to a particular architecture (e.g., machine instructions for a particular processor). When the kernel must perform processor-specific operations, such as manipulating the contents of a processor cache, control passes to the architecture-specific code that was integrated into the kernel at compile time.[33] Although Linux relies on architecture-specific

©1981–2004, Deitel & Associates, Inc. All rights reserved.

code to control computer hardware, Linux may also be executed on a set of virtual hardware devices. The mini case study, User-Mode Linux (UML), describes one such Linux port.

For a system to execute properly on a particular architecture, the kernel must be ported to that architecture and compiled for a particular machine prior to execution. Likewise, applications may need to be compiled (and sometimes redesigned) to properly operate on a particular system. For many platforms, this work has already been accomplished—a variety of platform-specific distributions provide ports of common applications and system services.[34]

Mini Case Study

User-Mode Linux (UML)

Kernel development is a complicated and error-prone process that can result in numerous bugs. Unlike other software, the kernel may execute privileged instructions, meaning that a flaw in the kernel could damage a system's data and hardware. As a result, kernel development can be a tedious (and risky) endeavor. **User-Mode Linux (UML)** facilitates kernel development by allowing developers to test and debug the kernel without damaging the system on which it runs.

User-Mode Linux (UML) is a version of the Linux kernel that runs as a user application on a computer running Linux. Unlike most versions of Linux, which contain architecture-specific code to control devices, UML performs all architecture-specific operations using system calls to the Linux system on which it runs. As a result, UML is interestingly considered to be port of Linux to itself.[40]

The UML kernel runs in user mode, so it cannot execute privileged instructions available to the host kernel. Instead of controlling physical resources, the UML kernel creates virtual devices (represented as files on the host system) that simulate real devices. Because the UML kernel does not control any real hardware, it cannot damage the system.

Almost all kernel mechanisms, such as process scheduling and memory management, are handled in the UML kernel; the host kernel executes only when privileged access to hardware is required. When a UML process issues a system call, the UML kernel intercepts and handles it before it can be sent to the host system. Although this technique incurs significant overhead, UML's primary goal is to provide a safe (i.e., protected) environment in which to execute software, not to provide high performance.[41]

The UML kernel has been applied to more than just testing and debugging. For example, UML can be used to run multiple instances of Linux at once. It can also be used to port Linux such that it runs as an application in operating systems other than Linux. This could allow users to run Linux on top of a UNIX or a Windows system. The Web Resources section at the end of this chapter provides a link to a Web site that documents UML usage and development.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.4.2 Loadable Kernel Modules

Adding functionality to the Linux kernel, such as support for a particular file system or a new device driver, can be tedious. Because the kernel is monolithic, drivers and file systems are implemented in kernel space. Consequently, to permanently add support for a device driver, users must patch the kernel source by adding the driver code, then recompiling the kernel. This can be a lengthy and error-prone process, so an alternative method has been developed for adding features to the kernel—**loadable kernel modules**.

A kernel module contains object code that, when loaded, is dynamically linked to a running kernel (see Section 2.8, Compiling, Linking and Loading). If a device driver or file system is implemented as a loadable kernel module, it can be loaded into the kernel on demand (i.e., when first accessed) without any additional kernel configuration or compilation. Also, because modules can be loaded on demand, moving code from the kernel into modules reduces the **memory footprint** of the kernel; hardware and file system drivers are not loaded into memory until needed. Modules execute in kernel mode (as opposed to user mode) so they can access kernel functions and data structures. Consequently, loading an improperly coded module can lead to disastrous effects in a system, such as data corruption.[35]

When a module is loaded, the module loader must resolve all references to kernel functions and data structures. Kernel code allows modules to access functions and data structures by exporting their names to a symbol table.[36] Each entry in the symbol table contains the name and address of a kernel function or data structure. The module loader uses the symbol table to resolve references to kernel code.[37]

Because modules execute in kernel mode, they require access to symbols in the kernel symbol table, which allows modules to access kernel functions. However, consider what can happen if a function is modified between kernel versions. If a module was written for a prior kernel version, the module may expect a particular, yet invalid, result from a current kernel function (e.g., an integer value instead of an unsigned long value), which may in turn lead to errors such as exceptions. To avoid this problem, the kernel prevents users from loading modules written for a version of the kernel other than the current one, unless explicitly overridden by the superuser.[38]

Modules must be loaded into the kernel before use. For convenience, the kernel supports dynamic module loading. When compiling the kernel, the user is given the option to enable or disable ***kmod***—a kernel subsystem that manages modules without user intervention. The first time the kernel requires access to a module, it issues a request to *kmod* to load the module. *Kmod* determines any module dependencies, then loads the requested module. If a requested module depends on other modules that have not been loaded, *kmod* will load those modules on demand.[39]

20.5 Process Management

The process management subsystem is essential to providing efficient multiprogramming in Linux. Although responsible primarily for allocating processors to processes,

©1981–2004, Deitel & Associates, Inc. All rights reserved.

the process management subsystem also delivers signals, loads kernel modules and receives interrupts. The process management subsystem contains the **process scheduler**, which provides processes access to a processor in a reasonable amount of time.

20.5.1 Process and Thread Organization

In Linux systems, both processes and threads are called **tasks**; internally, they are represented by a single data structure. In this section, we distinguish processes from threads from tasks where appropriate. The process manager maintains a list of all tasks using two data structures. The first is a circular, doubly linked list in which each entry contains pointers to the previous and next tasks in the list. This structure is accessed when the kernel must examine all tasks in the system. The second is a hash table. When a task is created, it is assigned a unique **PID (process identifier)**. Process identifiers are passed to a hash function to determine their location in the process table. The hashing method provides quick access to a specific task's data structure when the kernel knows its PID.[42]

Each task in the process table is represented by a `task_struct` structure, which serves as the process descriptor (i.e., the PCB). The `task_struct` structure stores variables and nested structures containing information describing a process. For example, the variable `state` stores information about the current task state. [*Note*: The kernel is primarily written using the C programming language and makes extensive use of structures to represent software entities.]

A task transitions to the *running* state when it is dispatched to a processor (Fig. 20.2). A task enters the *sleeping* state when it blocks and the *stopped* state when it is suspended. The *zombie* state indicates that a task has been terminated but has not yet been removed from the system. For example, if a process contains several threads, it will enter the *zombie* state until its threads have been notified that it received a termination signal. A task in the *dead* state may be removed from the system. The states *active* and *expired* are process scheduling states (described in the next section), which are not stored in the variable `state`.

Other important task-specific variables permit the scheduler to determine when a task should run on a processor. These variables include the task's priority, whether the task is a real-time task and, if so, which real-time scheduling algorithm should be used (real-time scheduling is discussed in the next section).[43]

Nested structures within a `task_struct` store additional information about a task. One such structure, `mm_struct`, describes the memory allocated to a task (e.g., the location of its page table in memory and the number of tasks sharing its address space). Additional structures nested within a `task_struct` contain information such as register values that store a task's execution context, signal handlers and the task's access rights.[44] These structures are accessed by several kernel subsystems other than the process manager.

When the kernel is booted, it typically loads a process called ***init***, which then uses the kernel to create all other tasks.[45] Tasks are created using the `clone` system call; any calls to `fork` or `vfork` are converted to clone system calls at compile

©1981–2004, Deitel & Associates, Inc. All rights reserved.

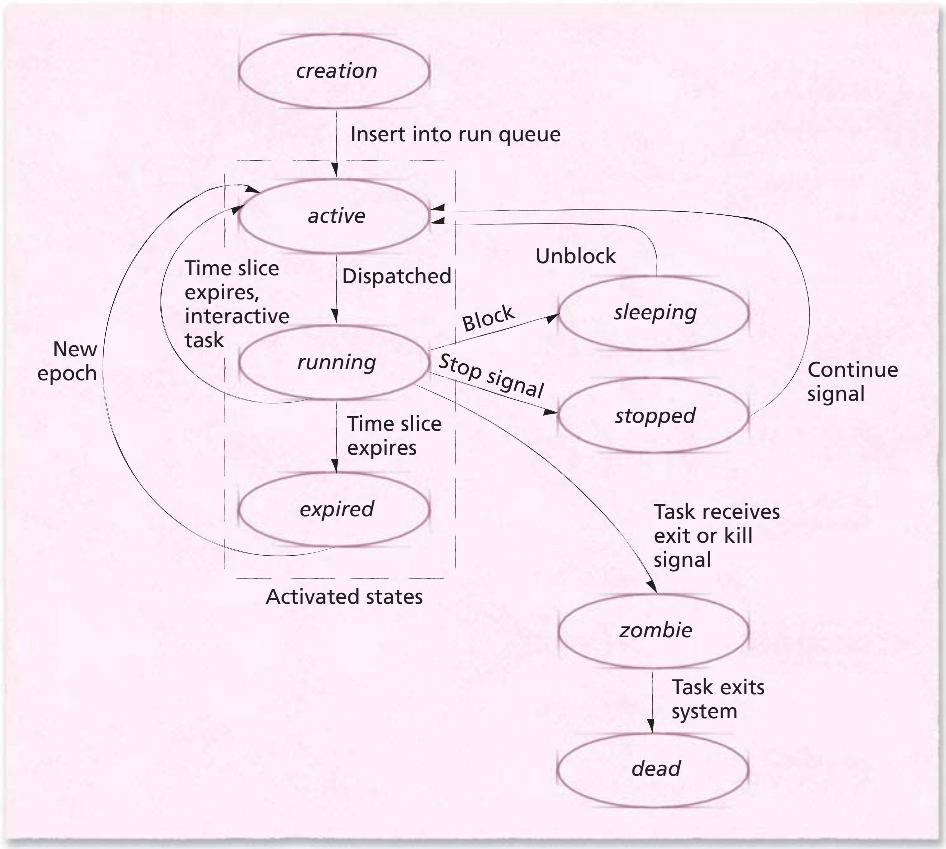


Figure 20.2 | Task state-transition diagram.

time. The purpose of `fork` is to create a child task whose virtual memory space is allocated using copy-on-write to improve performance (see Section 10.4.6, Sharing in a Paging System). When the child or the parent attempts to write to a page in memory, the writer is allocated its own copy of the page. As discussed in Section 10.4.6, copy-on-write can lead to poor performance if a process calls `execve` to load a new program immediately after the `fork`. For example, if the parent executes before its child, a copy-on-write will be performed for any page the parent modifies. Because the child will not use any of its parent's pages (if the child will immediately call `execve` when it executes), this operation is pure overhead. Therefore, Linux supports the `vfork` call, which improves performance when child processes will call `execve`. `vfork` suspends the parent process until the child calls `execve` or `exit`, to ensure that the child loads its new pages before the parent causes any wasteful copy-on-write operations. `vfork` further improves performance by not copying the parent's page tables to the child, because new page table entries will be created when the child calls `execve`.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Linux Threads and the `clone` System Call

Linux provides support for threads using the `clone` system call, which enables the calling process to specify whether the thread shares the process's virtual memory, file system information, file descriptors and/or signal handlers.[46] In Section 4.2, Definition of Thread, we mentioned that processor registers, the stack and other thread-specific data (TSD) are local to each thread, while the address space and open file handles are global to the process that contains the threads. Thus, depending on how many of the process's resources are shared with its thread, the resulting thread may be quite different from the threads described in Chapter 4.

Linux's implementation of threads has generated much discussion regarding the definition of a thread. Although `clone` creates threads, they do not precisely conform to the POSIX thread specification (see Section 4.8, POSIX and Pthreads). For example, two or more threads that were created using a `clone` call specifying maximum resource sharing still maintain several data structures that are not shared with all threads in the process, such as access rights.[47]

When `clone` is called from a kernel process (i.e., a process that executes kernel code), it creates a **kernel thread** that differs from other threads in that it directly accesses the kernel's address space. Several daemons within the kernel are implemented as kernel threads—these daemons are services that sleep until awakened by the kernel to perform tasks such as flushing pages to secondary storage and scheduling software interrupts (see Section 20.8.6, Interrupts).[48] These tasks are generally maintenance related and execute periodically.

There are several benefits to the Linux thread implementation. For example, Linux threads simplify kernel code and reduce overhead by requiring only a single copy of task management data structures.[49] Moreover, although Linux threads are less portable than POSIX threads, they allow programmers the flexibility to tightly control shared resources between tasks. A recent Linux project, Native POSIX Thread Library (NPTL), has achieved nearly complete POSIX conformance and is likely to become the default threading library in future Linux distributions.[50]

20.5.2 Process Scheduling

The goal of the Linux process scheduler is to run all tasks within a reasonable amount of time while respecting task priorities, maintaining high resource utilization and throughput, and reducing the overhead of scheduling operations. The process scheduler also addresses Linux's role in the high-end computer system market by scaling to SMP and NUMA architectures while providing high processor affinity. One of the more significant scalability enhancements in version 2.6 is that all scheduling functions are constant-time operations, meaning that the time required to execute scheduling functions does not depend on the number of tasks in the system.[51]

At each system timer interrupt (an architecture-specific number set to 1 millisecond by default for the IA-32 architecture[52]), the kernel updates various bookkeeping data structures (e.g., the amount of time a task has been executing) and performs scheduling operations as necessary. Because the scheduler is preemptive,

©1981–2004, Deitel & Associates, Inc. All rights reserved.

each task runs until its quantum, or time slice, expires, a higher priority process becomes runnable or the process blocks. Each task's time slice is calculated as a function of the process's priority upon release of the processor (with the exception of real-time tasks, which are discussed later). To prevent time slices from being too small to allow productive work or so large as to diminish response times, the scheduler ensures that the time slice assigned to each task is between 10 and 200 timer intervals, corresponding to a range of 10–200 milliseconds on most systems (like most scheduler parameters, these default values have been chosen empirically). When a task is preempted, the scheduler saves the task state to its `task_struct` structure. If the process's time slice has expired, the scheduler recalculates the process's priority, determines the task's next time slice and dispatches the next process.

Run Queues

Once a task has been created using `clone`, it is placed in a processor's **run queue**, which contains references to all tasks competing for execution on that processor. Run queues, similar to multilevel feedback queues (Section 8.7.6, Multilevel Feedback Queues), assign tasks to priority levels. The **priority array** maintains pointers to each level of the run queue. Each entry in the priority array points to a list of tasks—a task of priority *i* is placed in the *i*th entry of a priority array in the run queue (Fig. 20.3).

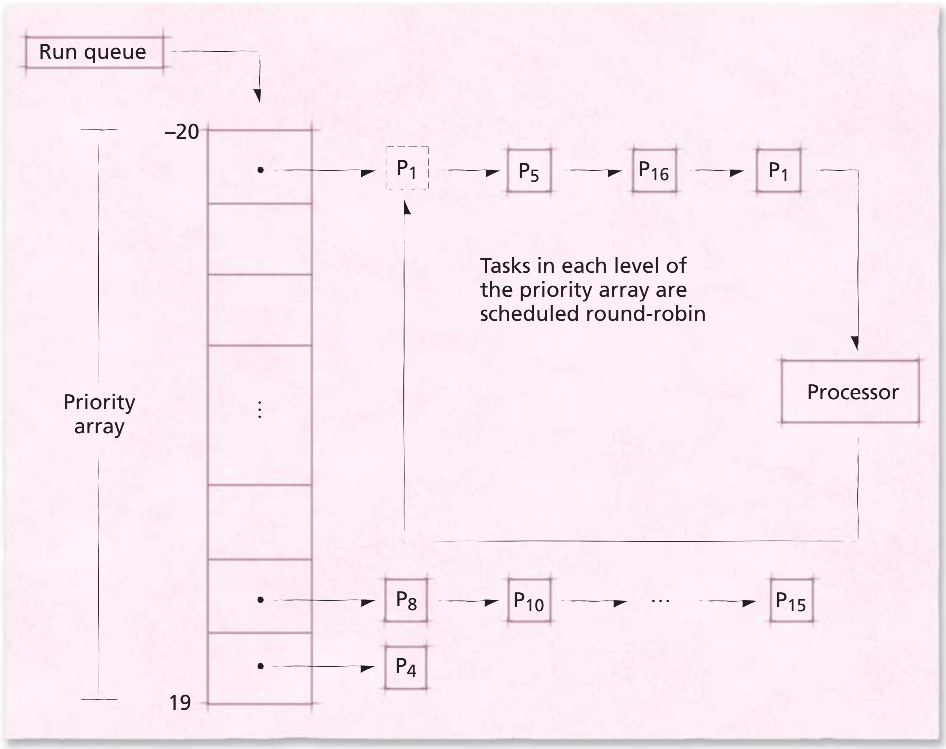


Figure 20.3 | Scheduler priority array.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

The scheduler dispatches the task at the front of the list in the highest level of the priority array. If more than one task exists in a level of the priority array, tasks are dispatched from the priority array round-robin. When a task enters the *blocked* or *sleeping* (i.e., *waiting*) state, or is otherwise unable to execute, that task is removed from its run queue.

One goal of the scheduler is to prevent indefinite postponement by defining a period of time called an **epoch** during which each task in the run queue will execute at least once. To distinguish processes that are considered for processor time from those that must wait until the next epoch, the scheduler defines an ***active* state** and an ***expired* state**. The scheduler dispatches only processes in the *active* state.

The duration of an epoch is determined by the **starvation limit**—an empirically derived value that provides high-priority tasks with good response times while ensuring that low-priority tasks are dispatched often enough to perform productive work within a reasonable amount of time. By default, the starvation limit is set to $10n$ seconds, where n is the number of tasks in the run queue. When the current epoch has lasted longer than the starvation limit, the scheduler transitions each active task in the run queue to the *expired* state (the transition occurs after each active task's time slice expires). This temporarily suspends high-priority tasks (with the exception of real-time tasks), allowing low-priority tasks to execute. When all tasks in the run queue have executed at least once, all tasks in that run queue will be in the *expired* state. At this point, the scheduler transitions all tasks in the run queue to the *active* state and a new epoch begins.[53]

To simplify the transition from the *expired* state to the *active* state at the end of an epoch, the Linux scheduler maintains two priority arrays for each processor. The priority array that contains tasks in the active state is called the **active list**. The priority array that stores expired tasks (i.e., tasks that are not allowed to execute until the next epoch) is called the **inactive** (or **expired**) **list**. When a task transitions from the *active* state to the *expired* state, it is placed in the level of the expired list's priority array corresponding to its priority when it transitioned to the *expired* state. At the end of an epoch, all tasks are located in the *expired* state and must transition to the *active* state. The scheduler performs this operation quickly by simply swapping the pointers to the expired list and the active list. By maintaining two priority arrays per process, the scheduler can transition all tasks in a run queue using a single swap operation, a performance enhancement that generally outweighs the nominal memory overhead due.[54]

The Linux scheduler scales to multiprocessor systems by maintaining one run queue for each physical processor in the system. One reason for per-processor run queues is to assign tasks to execute on particular processors to exploit processor affinity. Recall from Chapter 15 that processes in some multiprocessor architectures, such as NUMA, achieve higher performance when a task's data is stored in a processor's local memory and in a processor's cache. Consequently, tasks can achieve higher performance if they are consistently assigned to a single processor (or node). However, per-processor run queues risk unbalancing processor loads, leading to

©1981–2004, Deitel & Associates, Inc. All rights reserved.

reduced system performance and throughput (see Section 15.7, Process Migration). Later in this section, we discuss how Linux addresses this issue by dynamically balancing the number of tasks executing on each processor in the system.

Scheduling Priority

In the Linux scheduler, a task's priority affects the size of its time slice and the order in which it executes on a processor. Upon creation, tasks are assigned a **static priority**, also called the **nice value**. The scheduler recognizes 40 distinct priority levels, ranging from –20 to 19. Conforming to UNIX convention, smaller priority values denote higher priority in the scheduling algorithm (i.e., –20 is the highest priority a process can attain).

One goal of the Linux scheduler is to provide a high level of system interactivity. Because interactive tasks typically block to perform I/O or sleep (e.g., while waiting for a user response), the scheduler dynamically boosts the priority (by decrementing the static priority value) of a task that yields its processor before the task's time slice expires. This is acceptable because I/O-bound processes normally use the processor only briefly before generating an I/O request. Thus, giving I/O-bound tasks high priority has little effect on processor-bound tasks, which might use the processor for hours at a time if the system makes the processor available on a nonpreemptible basis. The modified priority level is called a task's **effective priority**, which is calculated when a task sleeps or consumes its time slice. A task's effective priority determines the level of the priority array in which a task is placed. Therefore, a task that receives a priority boost is placed in a lower level of the priority array, meaning it will execute before tasks of a higher effective priority value.

To further improve interactivity, the scheduler penalizes a processor-bound task by increasing its static priority value. This places a processor-bound task in a higher level of the priority array, meaning tasks of a smaller effective priority will be executed before it. Again, this ultimately has little effect on processor-bound tasks because the higher-priority interactive tasks execute only briefly before blocking.

To ensure that a task executes at or near the priority it was initially assigned, the task scheduler does not allow a task's effective priority to differ from its static priority by more than five units. In this sense, the scheduler honors the priority levels assigned to a task when it was created.

Scheduling Operations

The scheduler removes a task from a processor if the task is interrupted, preempted (e.g., if its time slice expires) or blocks. Each time a task is removed from a processor, the scheduler calculates a new time slice. If the task blocks, or is otherwise unable to execute, it is **deactivated**, meaning that it is removed from the run queue until it becomes ready to execute. Otherwise, the scheduler determines whether the task should be placed in the active list or the inactive list. The algorithm that determines this has been empirically derived to provide good performance; its primary factors are a task's static and effective priorities.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

The result of the algorithm is depicted in Fig. 20.4. The *y*-axis of Fig. 20.4 indicates a task's static priority value and the *x*-axis represents a task's priority adjustment (i.e., boost or penalty) The shaded region indicates sets of static priority values and priority adjustments that cause a task to be rescheduled, meaning that it is placed at the end of its corresponding priority array in the active list. In general, if a task is of high priority and/or has received a significant bonus to its effective priority, it is rescheduled. This allows high-priority, I/O-bound and interactive tasks to execute more than once per epoch. In the unshaded region, tasks that are of low priority and/or have received priority penalties are placed in the expired list.

When a user process `clones`, it may seem reasonable to allocate each child its own time slice. However, if a task spawns a large number of new tasks, and all of its children are allocated their own time slices, other tasks in the system might experience unreasonably poor response times during that epoch. To improve fairness, Linux requires that each parent process initially share its time slice with its children when a user process `clones`. The scheduler enforces this requirement by assigning half of the parent's original time slice to both the parent process and its child the child is spawned. To prevent legitimate processes from suffering low levels of ser-

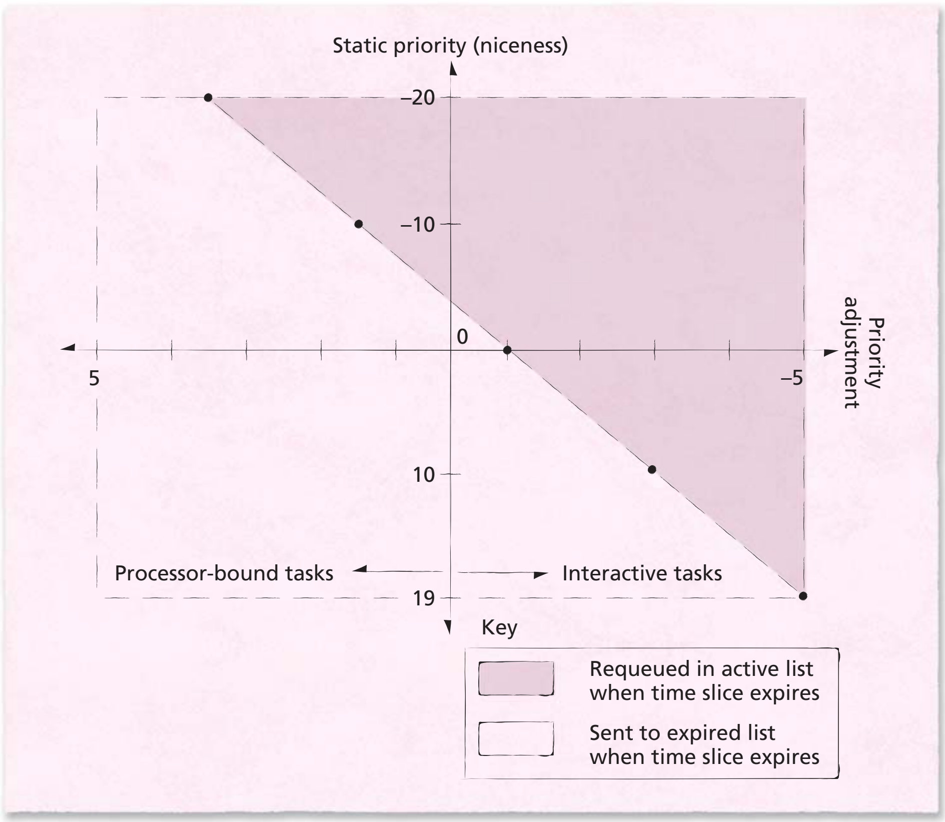


Figure 20.4 | *Priority values and their corresponding levels of interactivity.*[55]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

vice due to spawning child processes, this reduced time slice applies only during the remainder of the epoch during which the child is spawned.

Multiprocessor Scheduling

Because the process scheduler maintains tasks in a per-processor run queue, tasks will generally exhibit high processor affinity. This means that a task will likely be dispatched to the same processor for each of its time slices, which can increase performance when a task's data and instructions are located in a processor's caches. However, such a scheme could allow one or several processors on an SMP system to lie idle even during a heavy system load. To avoid this, if the scheduler detects that a processor is idle, it performs **load balancing** to migrate tasks from one processor to another to improve resource utilization. If the system contains only one processor, load balancing routines are removed from the kernel when it is compiled.

The scheduler determines if it should perform load balancing routines after each timer interrupt, which is set to one millisecond on IA-32 systems. If the processor that issued the timer interrupt is idle (i.e., its run queue is empty), the scheduler attempts to migrate tasks from the processor with the heaviest load (i.e., the processor that contains the largest number of processes in its run queue) to the idle processor. To reduce load balancing overhead, if the processor that triggered the interrupt is not idle, the scheduler will attempt to move tasks to that processor every 200 timer interrupts instead of after every timer interrupt.[56]

The scheduler determines processor load by using the average length of each run queue over the past several timer interrupts, to minimize the effect of variations in processor loads on the load balancing algorithm. Because processor loads tend to change rapidly, the goal of load balancing is not to adjust the size of two run queues until they are of equal length; rather, it is to reduce the imbalance between the number of tasks in each run queue. As a result, tasks are removed from the larger run queue until the difference in size between the two run queues has been halved. To reduce overhead, load balancing is not performed unless the run queue with the heaviest load contains 25 percent more tasks than the run queue of the processor performing the load balancing.[57]

When the scheduler selects tasks for balancing, it attempts to choose tasks whose performance will be least affected by moving from one processor to another. In general, the least-recently active task on a processor will most likely be cache-cold on the processor—a **cache-cold** task does not contain much (or any) of the task's data in its processor's cache, whereas a **cache-hot** task contains most (or all) of the task's data in the processor cache. Therefore, the scheduler chooses to migrate tasks that are most likely cache-cold (i.e., tasks that have not executed recently).

Real-Time Scheduling

The scheduler supports soft real-time scheduling by attempting to minimize the time during which a real-time task waits to be dispatched to a processor. Unlike a normal task, which is eventually placed in the expired list to prevent low-priority tasks from being indefinitely postponed, a real-time task is always placed in the

©1981–2004, Deitel & Associates, Inc. All rights reserved.

active list after its quantum expires. Further, real-time tasks always execute with higher priority than normal tasks. Because the scheduler always dispatches a task from the highest-priority queue in the active list (and real-time tasks are never removed from the active list), normal tasks cannot preempt real-time tasks.

The scheduler complies with the POSIX specification for real-time processes by allowing real-time tasks to be scheduled using the default scheduling algorithm described in the previous sections, or using the round-robin or FIFO scheduling algorithms. If a task specifies round-robin scheduling and its time slice has expired, the task is allocated a new time slice and is enqueued at the end of its priority array in the active list. If the task specifies FIFO scheduling, it is not assigned a time slice and therefore will execute on a processor until it exits, sleeps, blocks or is interrupted.[58] Clearly, real-time processes can indefinitely postpone other processes if coded improperly, resulting in poor response times. To prevent accidental or malicious misuse of real-time tasks, only users with root privileges can create them.

20.6 Memory Management

During the development of kernel versions 2.4 and 2.6, the memory manager was heavily modified to improve performance and scalability. The memory manager supports both 32- and 64-bit addresses as well as nonuniform memory access (NUMA) architectures to allow it to scale from desktop computers and workstations to servers and supercomputers.

20.6.1 Memory Organization

On most architectures, a system's physical memory is divided into fixed-size page frames. Generally, Linux allocates memory using a single page size (often 4KB or 8KB); on some architectures that support large pages (e.g., 4MB), kernel code may be placed in large pages. This can improve performance by minimizing the number of entries for kernel page frames in the translation lookaside buffer (TLB).[59] The kernel stores information about each page frame in a `page` structure. This structure contains variables that describe page usage, such as the number of processes sharing the page and flags indicating the state of the page (e.g., dirty, unused, etc).[60]

Virtual Memory Organization

On 32-bit systems, each process can address 2^{32} bytes, meaning that each virtual address space is 4GB. The kernel supports larger virtual address spaces on 64-bit systems—up to 2 petabytes (i.e., 2 million gigabytes) on Intel Itanium processors (the Itanium processor uses only 51 bits to address main memory, but the IA-64 architecture can support up to 64-bit physical addresses).[61] In this section, we focus on the 32-bit implementation of the virtual memory manager. Entries describing the virtual-to-physical address mappings are located in each process's page tables. The virtual memory system supports up to three levels of page tables to locate the mappings between virtual pages and page frames (see Fig. 20.5). The first level of the page table hierarchy, called the **page global directory**, stores addresses of sec-

©1981–2004, Deitel & Associates, Inc. All rights reserved.

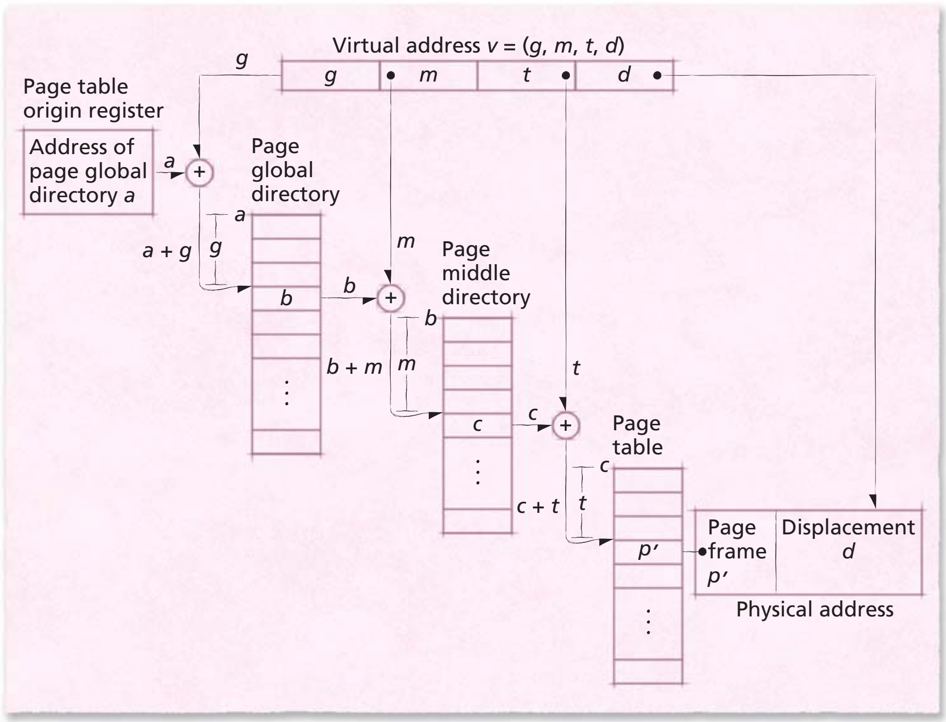


Figure 20.5 | Page table organization.

ond-level tables. Second-level tables, called **page middle directories**, store addresses of the third-level tables. The third level, simply called page tables, maps virtual pages to page frames.

The kernel partitions a virtual address into four fields to provide processors with multilevel page address translation information. The first three fields are indices into the process's page global directory, page middle directory and page table, respectively. These three fields allow the system to locate the page frame corresponding to a virtual page. The fourth field contains the displacement (also called offset) from the physical address of the beginning of the page frame.[69] Linux disables page middle directories when running on the IA-32 architecture, which supports only two levels of page tables when the Physical Address Extension (PAE) feature is disabled. Page middle directories are enabled for 64-bit architectures that support three or more levels of page tables (e.g., the x86-64 architecture, which supports four levels of page tables).

Virtual Memory Areas

Although a process's virtual address space is composed of individual pages, the kernel uses a higher-level mechanism, called **virtual memory areas**, to organize the virtual memory a process is using. A virtual memory area describes a contiguous set of pages in a process's virtual address space that are assigned the same protection

©1981–2004, Deitel & Associates, Inc. All rights reserved.

(e.g., read-only, read/write, executable) and backing store. The kernel stores a process's executable code, heap, stack and each memory-mapped file (see Section 13.9, Data Access Techniques) in separate virtual memory areas.[70, 71]

When a process requests additional memory, the kernel attempts to satisfy that request by enlarging an existing virtual memory area. The virtual memory area the kernel selects depends on the type of memory the process is requesting (e.g., executable code, heap, stack, etc.). If the process requests memory that does not correspond to an existing virtual memory area, or if the kernel cannot allocate a contiguous address space of the requested size in an existing virtual memory area, the kernel creates a new virtual memory area.[72]

Virtual Memory Organization for the IA-32 Architecture

In Linux, virtual memory organization is architecture specific. In this section, we discuss how the kernel organizes virtual memory by default to optimize performance on the IA-32 architecture.

When the kernel performs a context switch, it must provide the processor with page address translation information for the process that is about to execute (see Section 10.4.1). Recall from Section 10.4.3 that an associative memory called the translation lookaside buffer (TLB) stores recently used page table entries (PTEs) so that the processor can quickly perform virtual-to-physical address translations for the process that is currently running. Because each process is allocated a different virtual address space, PTEs for one process are not valid for another. As a result, the PTEs in the TLB must be removed after a context switch. This is called flushing the TLB—the processor removes each PTE from the TLB and updates the PTEs in main memory to match any modified PTEs in the TLB. In particular, each time the value of the page table origin register changes, the TLB must be flushed. The overhead due to a TLB flush can be substantial because the processor must access main memory to update each PTE that is flushed. If the kernel changes the value of the page table origin register to execute each system call, the overhead due to TLB flushing can significantly reduce performance.

To reduce the number of expensive TLB flush operations, the kernel ensures that it can use any process's page table origin register to access the kernel's virtual address space. The kernel does this by dividing each process's virtual address space into user addresses and kernel addresses. The kernel allows each process to access up to 3GB of the process's virtual address space—the virtual addresses from zero to 3GB. Therefore, virtual-to-physical address translation information can vary between processes for the first 3GB of each 32-bit virtual address space. The kernel address space is the remaining 1GB of virtual addresses in each process's 32-bit virtual address space (addresses ranging from 3GB to 4GB), as shown in Fig. 20.6. The virtual-to-physical address translation information for this region of memory addresses does not vary from one process to another. Therefore, when a user process invokes the kernel, the processor does not need to flush the TLB, which improves performance by reducing the number of times the processor accesses main memory to update page table entries.[62]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Often, the kernel must access main memory on behalf of user processes (e.g., to perform I/O); therefore, it must be able to access every page frame in main memory. However, today's processors require that all memory references use virtual addresses to access memory (if virtual memory is enabled). As a result, the kernel generally must use virtual addresses to access page frames.

When using virtual addresses, the kernel must provide the processor with PTEs that map the kernel's virtual pages to the page frames it must access. The kernel could create these PTEs each time it accessed main memory; however, doing so would create significant overhead. Therefore, the kernel creates PTEs that map most of the pages in the kernel's virtual address space permanently to page frames in main memory. For example, the first page of the kernel's virtual address space always points to the first page frame in main memory; the 100th page of the kernel's virtual address space always points to the 100th page frame in main memory (Fig. 20.6).

Note that creating a PTE (i.e., mapping virtual page to a page frame) does not allocate a page frame to the kernel or a user process. For example, assume that page frame number 100 stores a process's virtual page *p*. When the kernel accesses virtual page number 100 in the kernel virtual address space, the processor maps the virtual page number to page frame number 100, which stores the contents of *p*. Thus, the kernel's virtual address space is used to access page frames that may be allocated to the kernel or user processes.

Ideally, the kernel would be able to create PTEs that permanently map to each page frame in memory. However, if a system contains more than 1GB of main memory, the kernel cannot create a permanent mapping to every page frame because it reserves only 1GB of each 4GB virtual address space for itself. For example, when the kernel performs I/O on behalf of a user process, it must be able to access the data using pages in its 1GB virtual address space. However, if a user pro-

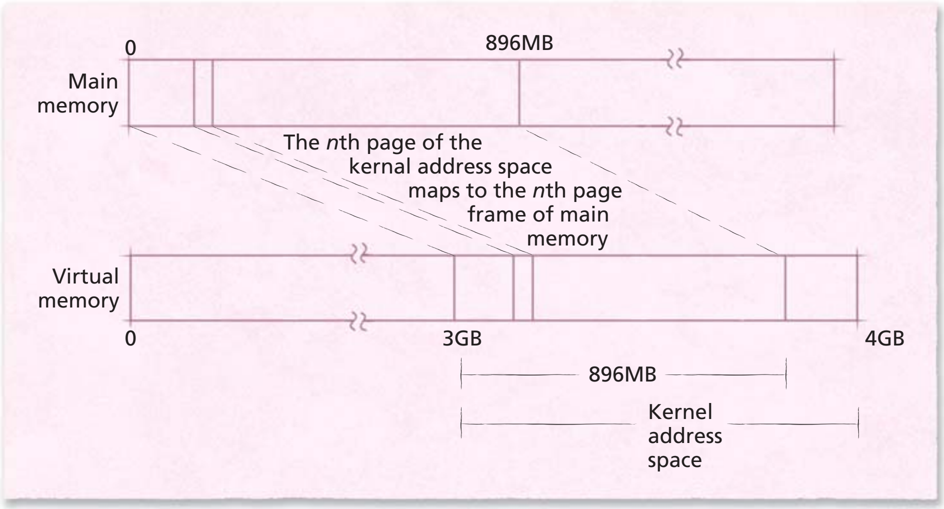

0
896MB
Main memory
The nth page of the kernal address space maps to the nth page frame of main memory
Virtual memory
0
3GB
4GB
896MB
Kernel address space

Figure 20.6 | *Kernel virtual address space mapping.*

©1981–2004, Deitel & Associates, Inc. All rights reserved.

cess requests I/O for a page that is stored at an address higher than 1GB, the kernel might not contain a mapping to that page. In this case, the kernel must be able to create a temporary mapping between a kernel virtual page and a user's physical page in main memory to perform the I/O. To address this problem, the kernel maps most of its virtual pages permanently to page frames and reserves several virtual pages to provide temporary mappings to the remaining page frames. In particular, the kernel creates PTEs that map the first 896MB of its virtual pages permanently to the first 896MB of main memory when the kernel boots. It reserves the remaining 128MB of its virtual address space for temporary buffers and caches that can be mapped to other regions of main memory. Therefore, if the kernel must access page frames beyond 896MB, it uses virtual addresses in this region to create a new PTE that temporarily maps a virtual page to a page frame.

Physical Memory Organization

The memory management system divides a system's physical address space into three **zones** (Fig. 20.7). The size of each zone is architecture dependent; in this section we present the configuration for the IA-32 architecture discussed in the previous section. The first zone, called **DMA memory**, includes the main memory locations from 0–16MB. The primary reason for creating a DMA memory zone is to ensure compatibility with legacy architectures. For example, some direct memory

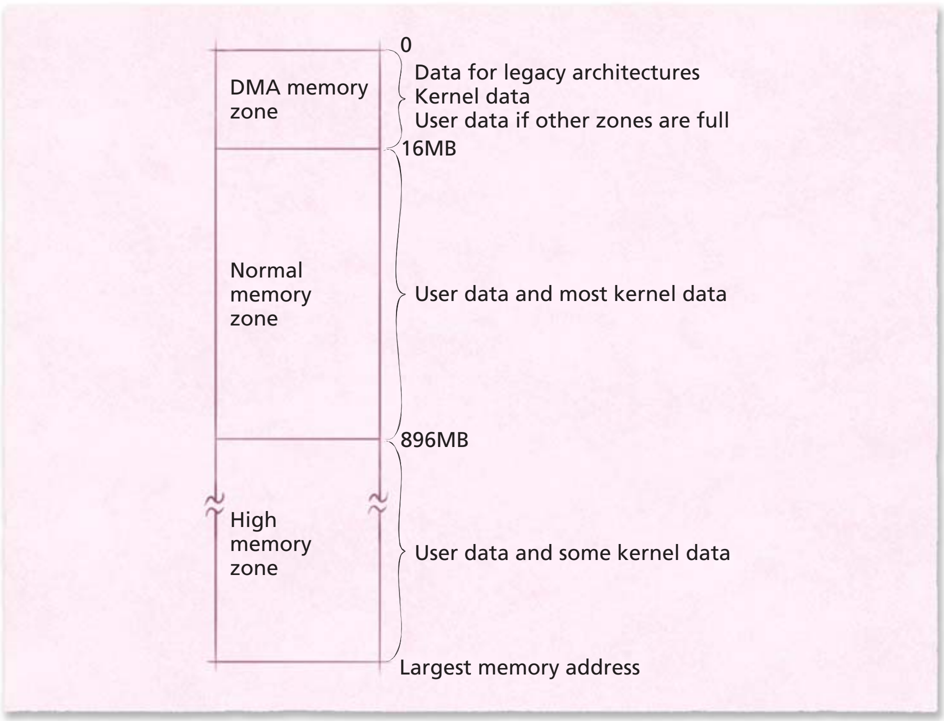

0
DMA memory zone
Data for legacy architectures
Kernel data
User data if other zones are full
16MB
Normal memory zone
User data and most kernel data
896MB
High memory zone
User data and some kernel data
Largest memory address

Figure 20.7 | Physical memory zones on the IA-32 Intel architecture.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

access (DMA) devices can address only up to 16MB of memory, so Linux reserves memory in this zone for such devices. The DMA memory zone also contains kernel data and instructions (e.g., bootstrapping code) and might be allocated for user processes if free memory is scarce.

The second zone, called **normal memory**, includes the physical memory locations between 16MB and up to 896MB. The normal memory zone can be used to store user and kernel pages as well as data from devices that can access memory greater than 16MB using DMA. Note that because the kernel's virtual address space is mapped directly to the first 896MB of main memory, most kernel data structures are located in the low memory zones (i.e., the DMA or normal memory zones). If these data structures were not located in these memory zones, the kernel could not provide a permanent virtual-to-physical address mapping for kernel data and might cause a page fault while executing its code. Page faults not only reduce kernel performance but can be fatal when performing error-handling routines.

The third zone, called **high memory**, includes physical memory locations from 896MB to a maximum of 64GB on Pentium processors. (Intel's Page Address Extension feature enables 36-bit memory addresses, allowing the system to access 2^{36} bytes, or 64GB, of main memory.) High memory is allocated to user processes, any devices that can access memory in this zone and temporary kernel data structures.[64]

Some devices, however, cannot access data in high memory because the number of physical addresses they can address is limited. In this case, the kernel copies such data to a buffer, called a **bounce buffer**, in DMA memory to perform I/O. After completing an I/O operation, the kernel copies any modified pages in the buffer to the page in high memory.[65, 66, 67]

Depending on the architecture, the first megabyte of main memory might contain data loaded into memory by initialization functions in the BIOS (see Section 2.3.1, Mainboard). To avoid overwriting such data, the Linux kernel code and data structures are loaded into a contiguous area of physical memory, typically beginning at the second megabyte of main memory. (The kernel reclaims most of the first megabyte of memory after loading.) Kernel pages are never swapped (i.e., paged) or relocated in physical memory. In addition to improving performance, the contiguous and static nature of kernel memory simplifies coding for kernel developers at a relatively low cost (the kernel footprint is approximately 2MB).[68]

20.6.2 Physical Memory Allocation and Deallocation

The kernel allocates page frames to processes using the **zone allocator**. The zone allocator attempts to allocate pages from the physical zone corresponding to each request. Recall that the kernel reserves as much of the DMA memory zone as possible for use by legacy architectures and kernel code. Also, performing I/O operations on high memory might require use of a bounce buffer, which is less efficient than using pages that are directly accessible by DMA hardware. Thus, although pages for user processes can be allocated from any zone, the kernel attempts to allocate them first from the high memory zone. If the high memory zone is full and

©1981–2004, Deitel & Associates, Inc. All rights reserved.

pages in normal memory are available, then the zone allocator uses pages from normal memory. Only when free memory is scarce in both the normal and the high zone of memory does the zone allocator select pages in DMA memory.[73]

When deciding which page frames to allocate, the zone allocator searches for empty pages in each zone's `free_area` vector. The `free_area` vector contains references to a zone's free lists and bitmaps that identify contiguous blocks of memory. Blocks of page frames are allocated in groups of powers of two; each element in a zone's `free_area` vector contains a list of blocks that are the same size—the *n*th element in the vector references a list of blocks of size 2^n.[74] Figure 20.8 illustrates the first three entries of the `free_area` vector.

To locate blocks of the requested size, the memory manager uses the **binary buddy algorithm** to search the `free_area` vector. The buddy algorithm, described by Knowlton and Knuth, is a simple physical page allocation algorithm that provides good performance.[75, 76] If there are no blocks of the requested size, a block of the next-closest size in the `free_area` vector is halved repeatedly until the resulting block is of the correct size. When the memory manager finds a block of the correct size, it allocates it to the process that requested it and places any orphaned free blocks in the appropriate lists.[77]

When memory is deallocated, the buddy algorithm groups contiguous free pages as follows. When a process frees a block of memory, the memory manager checks the bitmap (in the `free_area` vector) that tracks blocks of that size. If the bitmap indicates that an adjacent block is free, the memory manager combines the two blocks (buddies) into a larger block. The memory manager repeats this process until there are no blocks with which to combine the resulting block. The memory manager then inserts the block into the proper list in `free_area`.[78]

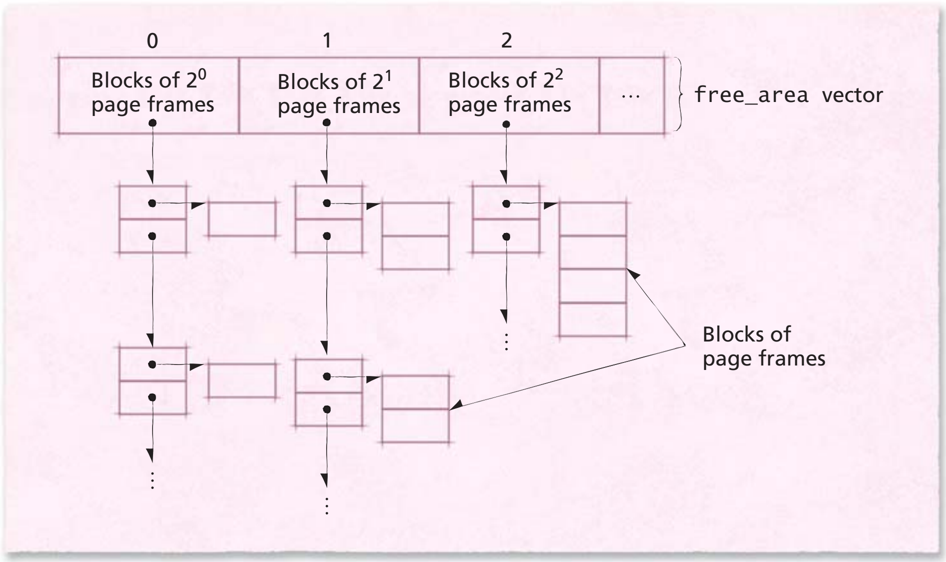

0
1
2
Blocks of 2^0 page frames
Blocks of 2^1 page frames
Blocks of 2^2 page frames
...
free_area vector
Blocks of page frames

Figure 20.8 | `free_area` *vector.*

©1981–2004, Deitel & Associates, Inc. All rights reserved.

There are several kernel data structures (e.g., the structure that describes virtual memory areas) that consume much less than a page of memory (4KB is the smallest page size on most systems). Processes tend to allocate and release many such data structures during the course of execution. Requests to the zone allocator to allocate such small amounts of memory would result in substantial internal fragmentation because the smallest unit of memory the zone allocator can supply is a page. Instead, the kernel satisfies such requests via the **slab allocator**. The slab allocator allocates memory from any one of a number of available **slab caches**.[79] A slab cache is composed of a number of objects, called slabs, that span one or more pages and contain structures of the same type. Typically, a **slab** is one page of memory that serves as a container for multiple data structures smaller than a page. When the kernel requests memory for a new structure, the slab allocator returns a portion of a slab in the slab cache for that structure. If all of the slabs in a cache are occupied, the slab allocator increases the size of the cache to include more slabs. These new slabs contain pages allocated using the appropriate zone allocator.[80]

As previously discussed, serious or fatal system errors can occur if the kernel causes a page fault during interrupt- or error-handling code. Similarly, a request to allocate memory while executing such code must not fail if the system contains few free pages. To prevent such a situation, Linux allows kernel threads and device drivers to allocate memory pools. A **memory pool** is a region of memory that the kernel guarantees will be available to a kernel thread or device driver regardless of how much memory is currently occupied. Clearly, extensive use of memory pools limits the number of page frames available to user processes. However, because a system failure could result from a failed memory allocation, the trade-off is justified.[81]

20.6.3 Page Replacement

The Linux memory manager determines which pages to keep in memory and which pages to replace (known as "swapping" in Linux) when free memory becomes scarce. Recall that only pages in the user region of a virtual address space can be replaced; most pages containing kernel code and data cannot be replaced.

As pages are read into memory, the kernel inserts them into the **page cache**. The page cache is designed to reduce the time spent performing disk I/O operations. When the kernel must flush (i.e., write) a page to disk, it does so through the page cache. To improve performance, the page cache employs write-back caching (see Section 12.8, Caching and Buffering) to clean dirty pages.[82]

Each page in the page cache must be associated with a secondary storage device (e.g., a disk) so the kernel knows where to place pages when they are swapped out. Pages that are mapped to files are associated with a file's inode, which describes the file's location on disk (see Section 20.7.1, Virtual File System, for a detailed description of inodes). As we discuss in the next section, pages that are not mapped to files are placed on secondary storage in a region called the system swap file.[83]

When physical memory is full and a nonresident page is requested by processes or the kernel, a page frame must be freed to fill the request. The memory

©1981–2004, Deitel & Associates, Inc. All rights reserved.

manager provides a simple, efficient page-replacement strategy. Figure 20.9 illustrates this strategy. In each memory zone, pages are divided into two groups: **active pages** and **inactive pages**. To be considered active, a page must have been referenced recently. One goal of the memory manager is to maintain the current working set inside the collection of active pages.[84]

Linux uses a variation of the clock page-replacement strategy (see Section 11.6.7). The memory manager uses two linked lists per zone to implement page replacement: the active list contains active pages, the inactive list contains inactive pages. The lists are organized such that the most-recently used pages are near the head of the active list, and the least-recently used pages are near the tail of the inactive list.[85]

When the memory manager allocates a page of memory to a process, the page's associated page structure is placed at the head of that zone's inactive list and

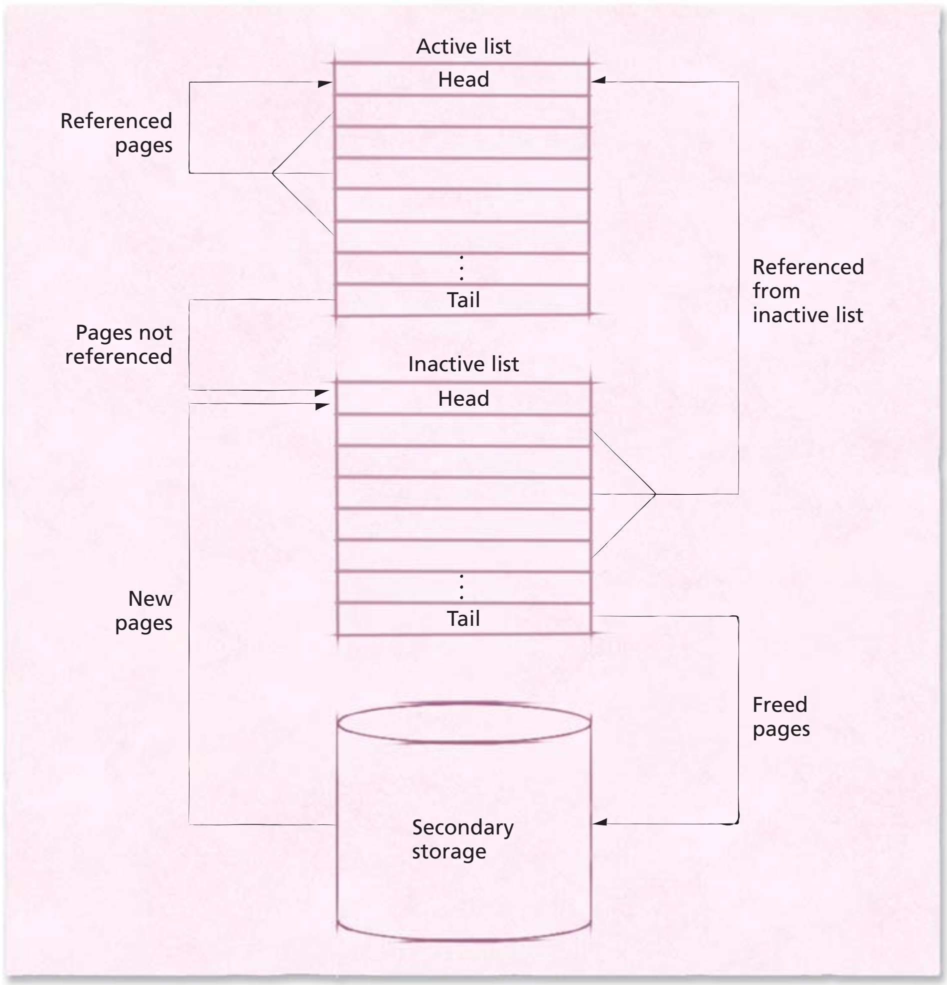


Figure 20.9 | *Page-replacement system overview.*

©1981–2004, Deitel & Associates, Inc. All rights reserved.

that page is marked as having been referenced by setting its referenced bit. The memory manager determines whether the page has been subsequently referenced at several points during kernel execution, such as when a PTE is flushed from the TLB. If the page has been referenced, the memory manager determines how to mark the page based on whether the page is active or inactive, and whether it has been referenced recently.

If the page is active or inactive and its referenced bit is off, the bit is turned on. Similar to the clock algorithm, this technique ensures that recently referenced pages are not selected for replacement. Otherwise, if the page is inactive and is being referenced for the second time (its referenced bit is already on), the memory manager moves the page to the head of the active list, then clears its referenced bit.[86] This allows the kernel to distinguish between referenced pages that have been accessed once and those that have been accessed more than once recently. The latter are placed in the active list so that they are not selected for replacement.

The memory manager updates the active list by transferring pages that have not been recently accessed to the inactive list. This is performed periodically and when available memory is low. The memory manager attempts to balance the lists such that approximately two-thirds of the total number of pages in the page cache are in the active list—an empirically derived value that achieves good performance in many environments.[87] The memory manager achieves this goal by periodically moving any unreferenced pages in the active list to the head of the inactive list.

This algorithm is repeated until the specified number of pages have been moved from the tail of the active list to the head of the inactive list. A page in the inactive list will remain in memory unless it is reclaimed (e.g., when free memory is low). While a page is in the active list, however, the page cannot be selected for replacement.[88]

20.6.4 Swapping

When available page frames become scarce, the kernel must decide which pages to swap out to free page frames for new requests. This is performed periodically by the kernel thread ***kswapd*** (the swap daemon), which reclaims pages by writing dirty pages to secondary storage. If the pages are mapped to a file in a particular file system (i.e., store file data in main memory), the system updates the file with any modifications to the page in memory. If the page corresponds to a process's data or procedure page, *kswapd* writes them to a region of data in secondary storage called the **system swap file**. *kswapd* selects pages to evict from entries at the tail of the inactive list.[89]

When swapping out a page, *kswapd* first determines whether it exists in the swap cache. The **swap cache** contains page table entries that describe whether a given page already exists in the system swap file on secondary storage. Under certain conditions, if the swap cache contains an entry for the page being swapped out, the page frame occupied by the page is freed immediately. By examining the swap cache, *kswapd* can avoid performing expensive I/O operations when an exact copy of the swapped page exists in the swap file.[90]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Before a page is replaced, the memory manager must determine whether to perform certain actions to ensure consistency (e.g., updating PTEs and writing data to disk). A page chosen for replacement cannot be immediately swapped under the following conditions:

- The page is shared (i.e., referenced by more than one process).
- The page has been modified.
- The page is **locked**—a process or device relies on its presence in main memory to perform an operation.[91]

If a page is being referenced by more than one process, *kswapd* must first **unmap** references to the page. The kernel unmaps a reference to a page by zeroing its PTE value. Linux uses **reverse mapping** to quickly find all page table entries pointing to a page, given a reference to a page frame. This is implemented in the `page` structure by a linked list of page table entries that reference the page. Without reverse mappings, the kernel would be required to search every page table in the system to find PTEs that map the page that is chosen for replacement. Although reverse mapping increases the size of each `page` object in the system, which in turn increases kernel memory usage, the performance improvement over searching page tables usually outweighs its cost.[92]

After the kernel unmaps all the page table entries that reference a page, it must determine if the page has been modified. Modified (i.e., dirty) pages must be flushed to disk before they can be freed. To improve performance and reduce data loss during system crashes, the kernel attempts to limit the number of dirty pages resident in memory. The kernel thread *pdflush* attempts to flush pages to disk (i.e., clean dirty pages) approximately every 5 seconds (depending on the system load) and defines an upper limit of 30 seconds during which pages can remain dirty. Once the disk flushing I/O is complete, *kswapd* can reclaim the page frame and allocate it to a new virtual page.[93, 94]

If a page is locked, *kswapd* cannot access the page to free it because a process or device relies on its presence in main memory to perform an operation. For example, a page of memory used to perform an I/O operation is typically locked. When the memory manager searches the inactive page list to choose pages for eviction, it does not consider locked pages. The page is freed on the next pass through the list if it is no longer locked and is still a member of the inactive list.[95]

20.7 File Systems

To meet the needs of users across multiple platforms, Linux must support a variety of file systems. When the kernel requires access to a specific file, it calls functions defined by the file system containing the file. Each particular file system determines how to store and access its data.

In Linux, a file refers to more than bits on secondary storage—files serve as access points to data, which can be found on a local disk, across a network, or even generated by the kernel itself. By abstracting the concept of a file, the kernel can

©1981–2004, Deitel & Associates, Inc. All rights reserved.

access hardware devices, interprocess communication mechanisms, data stored on disk and a variety of other data sources using a single generic file system interface. Developers use this interface to quickly add support for new file systems as they become available. Linux kernel version 2.6 includes support for more than 40 file systems that can be integrated into the kernel or loaded as modules.[96] These include general-purpose file systems (e.g., ext2, FAT and UDF), network file systems (e.g., NFS, CIFS and Coda) and file systems that exist exclusively in memory (e.g., procfs, sysfs, ramfs and tmpfs). In the sections that follow, we discuss the ext2, procfs, sysfs, ramfs and tmpfs file systems.

20.7.1 Virtual File System

Linux supports multiple file systems by providing a virtual file system (VFS) layer. The VFS abstracts the details of file access, allowing users to view all the files and directories in the system under a single directory tree. Users can access any file in the directory tree without knowledge of where, and under which file system, the file data are stored. All file-related requests are initially sent to the VFS layer, which provides an interface to access file data on any available file system. The VFS provides only a basic definition of the objects that comprise a file system. Individual file systems expand that basic definition to include details of how objects are stored and accessed.[97] Figure 20.10 illustrates this layered file system approach. Processes

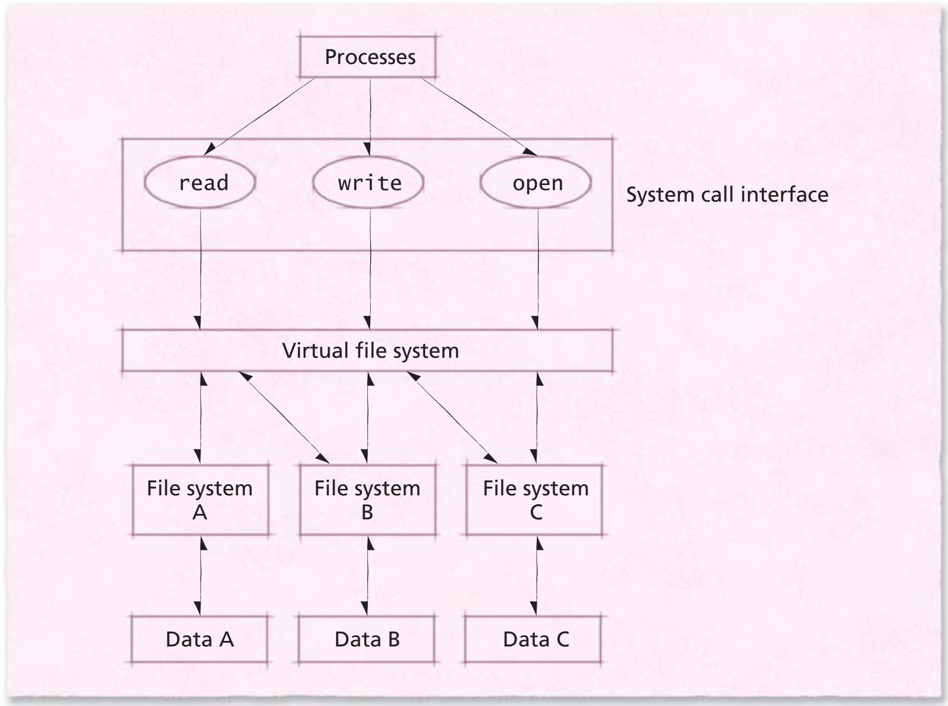


Figure 20.10 | Relationship between the VFS, file systems and data.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

issue system calls such as `read`, `write` and `open`, which are passed to the virtual file system. The VFS determines the file system to which the request corresponds and calls the corresponding routines in the file system driver, which perform the requested operations. This layered approach simplifies application programming and enables developers to add support for new file systems quickly, at the cost of nominal execution-time overhead.

The VFS uses files to read and write data that are not necessarily stored as bits on secondary storage. The virtual file system layer defines a number of objects that locate and provide access to data. One such object, called an **inode**, describes the location of each file, directory or link within every available file system. VFS inodes do not contain the name of the file they represent; rather, inodes are uniquely identified by a tuple containing an inode number (which is unique to a particular file system) and a number identifying the file system that contains the inode.[98] The VFS enables several file names to map to a single inode. This allows users to create hard links—multiple file names that map to the same inode within a file system.

Linux uses files to represent many objects, including named sets of data, hardware devices and shared memory regions. The broad usage of files originates in UNIX systems, from which Linux borrows many concepts.

The VFS represents each file using a file descriptor, which contains information about the inode being accessed, the position in the file being accessed and flags describing how the data is being accessed (e.g. read/write, append-only).[99] For clarity, we refer to VFS file objects as "file descriptors" and use the term "file" to refer to named data within a particular file system.

To map file descriptors to inodes, the VFS uses a **dentry (directory entry)** object. A dentry contains the name of the file or directory an inode represents. A file descriptor points to a dentry, which points to the corresponding inode.[100] Figure 20.11 shows a possible dentry representation of the `/home` directory and its contents. Each dentry contains a name and pointers to the dentry of its parent, children and siblings. For example, the dentry corresponding to `/home/chris` contains pointers to its parent (`/home`), children (`/home/chris/foo`, `/home/chris/bar` and `/home/chris/txt`) and sibling (`/home/jim`) directory entries. Using this information, the virtual file system can quickly resolve pathname-to-inode conversions. Dentries are discussed further in Section 20.7.2, Virtual File System Caches.

The Linux directory tree is comprised of one or more file systems, each comprised of a tree of inodes. When a file system is **mounted**, its contents are attached to a specified part of the primary directory tree. This allows processes to access data located in different file systems transparently via a single directory tree. A **VFS superblock** contains information about a mounted file system, such as the type of file system, its root inode's location on disk and housekeeping information that protects the integrity of the file system (e.g., the number of free blocks and free inodes in the system).[101] The VFS superblock is created by the kernel and resides exclusively in memory. Each file system must provide the VFS with superblock data when it is mounted.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

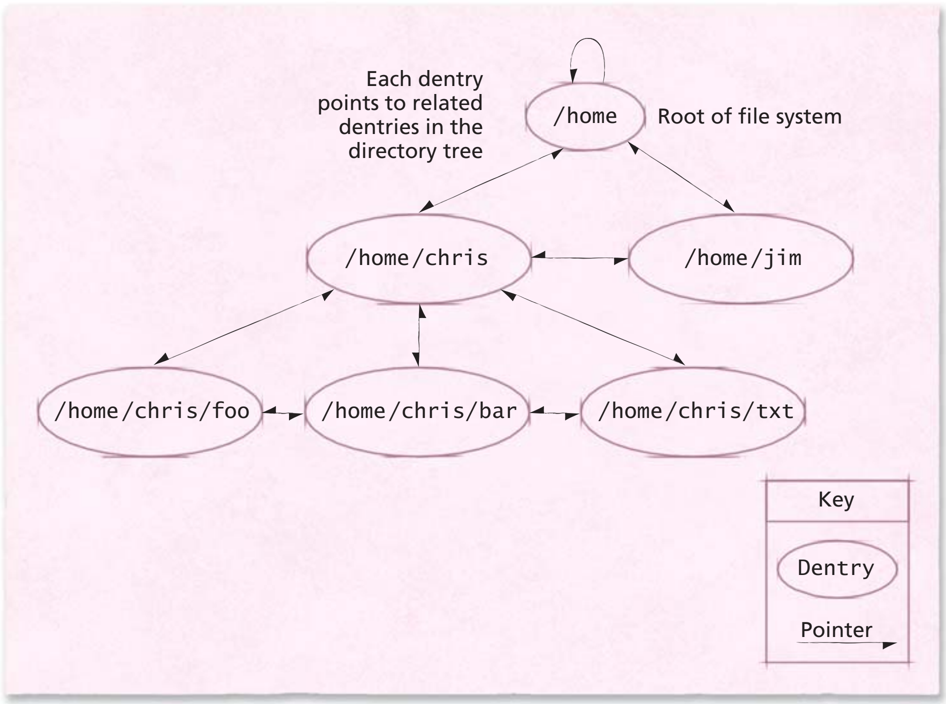


Figure 20.11 | *Dentry organization for a particular* /home *directory.*

The data stored in a file system's superblock is file-system dependent, but typically includes a pointer to the root inode (i.e., the inode that corresponds to the root of the file system) as well as information regarding the size and available space in the file system. Because a file system's superblock contains a pointer to the first inode in the file system, the operating system must load its superblock to access any other data in the file system. Most file systems place their superblock in one of the first blocks on secondary storage and maintain redundant copies of the superblock throughout their storage device to recover from damage.[102]

The virtual file system interprets data from the superblock, inodes, files and dentries to determine the contents of available file systems. The VFS defines generic file system operations and requires that each file system provide an implementation for each operation it supports. For example, the VFS defines a `read` function, but does not implement it. Example VFS file operations are listed in Fig. 20.12. Each file system driver must therefore implement a `read` function to allow processes to read its files. The virtual file system also provides generic file system primitives (e.g., files, superblocks and inodes). Each file system driver must assign file-system-specific information to these primitives.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

VFS operation	Intended use
`read`	Copy data from a file to a location in memory.
`write`	Write data from a location in memory to a file.
`open`	Locate the inode corresponding to a file.
`release`	Release the inode associated with a file. This can be performed only when all open file descriptors for that inode are closed.
`ioctl`	Perform a device-specific operation on a device (represented by an inode and file).
`lookup`	Resolve a pathname to a file system inode and return a dentry corresponding to it.

Figure 20.12 | VFS file and inode operations.

20.7.2 Virtual File System Caches

The virtual file system maintains a single directory tree composed of one or more file systems. To improve performance for file and directory access, the virtual file system maintains two caches—the **directory entry cache (dcache)** and the **inode cache**. These caches contain information about recently used entries in the directory tree. The cache entries represent objects in any available mounted file system.[103]

The dcache contains dentries corresponding to directories that have recently been accessed. This allows the kernel to quickly perform a pathname-to-inode translation if the file specified by the pathname is located in main memory. Because the amount of memory allocated to the dcache is limited, the VFS uses the dcache to store the most recently used dentries.[105] Although normally it cannot cache every file and directory in the system, the VFS ensures that if a dentry is in the dcache, its parent and other ancestors are also in the dcache. The only time this might not hold true is when file systems are accessed across a network (due to the fact that remote file system information can change without the local system being notified).[106]

Recall that when the VFS performs a pathname-to-inode translation, it uses dentries in the dcache to quickly locate inodes in the inode cache. The VFS then uses these inodes to locate a file's data when it is cached in main memory. Because the VFS relies on dentries to quickly locate inodes, each dentry's corresponding inode should be present in the inode cache. Therefore, the VFS ensures that each dentry in the dcache corresponds to an inode in the inode cache.

Conversely, if an inode is not referenced by a dentry, the VFS cannot access the inode. Therefore the VFS removes inodes that are no longer referenced by a dentry.[107, 108]

Locating the inode corresponding to a given pathname is a multistep process. The VFS must perform a directory-to-inode translation for each directory in the pathname. The translation begins at the root inode of the file system containing the pathname. The location of the file system's root inode is found in its superblock,

©1981–2004, Deitel & Associates, Inc. All rights reserved.

which is loaded into memory when the file system is mounted. Beginning at the root inode, the VFS must resolve each directory entry in the pathname to its corresponding inode.[109]

When searching for the inode that represents a given directory, the VFS first checks the dcache for the directory. If the dentry is found in the dcache, the corresponding inode must exist in the inode cache. Note that in Fig. 20.13 the dentry corresponding to `foo.txt` exists in the dentry cache; that dentry then points to an inode, which points to data in a file system.

If the VFS cannot find the dentry in the dcache, it searches for the inode directly in the inode cache.[110] Figure 20.13 illustrates this case using `link.txt`, a hard link to the file `bar.txt`. In this case, a process has previously referenced `link.txt`, meaning that a dentry corresponding to `link.txt` exists in the dcache. Because `link.txt` is a hard link to `bar.txt`, `link.txt`'s dentry points to `bar.txt`'s inode. When `bar.txt` is referenced for the first time, its dentry does not exist in the dentry cache. However, because a process has referenced `bar.txt` using a hard link, the inode corresponding to `bar.txt` exists in the inode cache. When the VFS does not find an entry for `bar.txt` in the dcache, it searches the inode cache and locates the inode corresponding to `bar.txt`.

If the dentry is not in the dcache and its corresponding inode is not in the inode cache, the VFS locates the inode by calling its parent directory inode's lookup function (which is defined by the underlying file system).[111] Once the directory is located, its associated inode and corresponding dentry are loaded into memory. The new inode is added to the inode cache and the dentry is added to the dcache.[112]

The VFS repeats the process of searching the caches before calling the lookup function for each directory in the pathname. By utilizing the caches, the VFS can

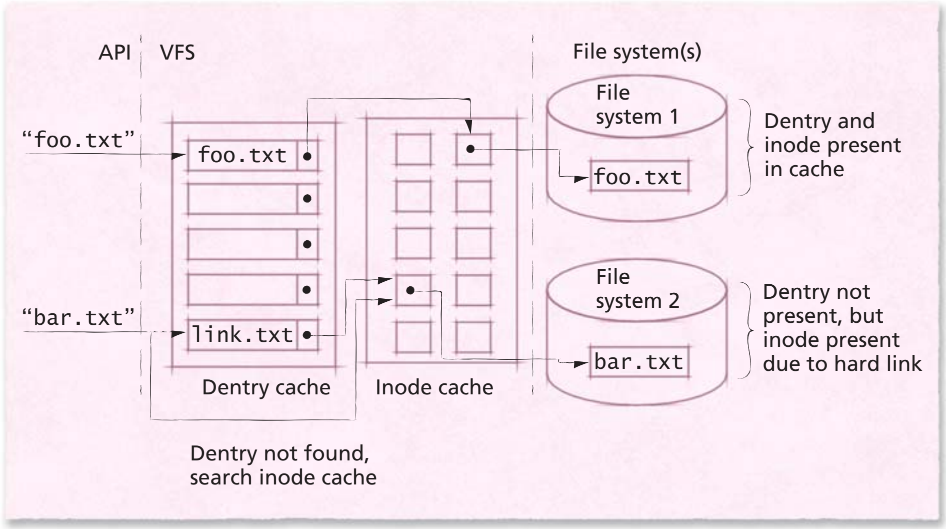

API
VFS
File system(s)
"foo.txt"
foo.txt
File system 1
foo.txt
Dentry and inode present in cache
"bar.txt"
link.txt
File system 2
bar.txt
Dentry not present, but inode present due to hard link
Dentry cache
Inode cache
Dentry not found, search inode cache

Figure 20.13 | Dentry and inode caches.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

avoid lengthy delays due to a file system's accessing inodes on disk, across a network, or from other media.

The inode `lookup` function is one of several functions that file systems typically implement (see Fig. 20.12). The primary responsibilities of the VFS are to cache file system data and pass file access requests to the appropriate file systems. Most file systems provide the VFS with their own implementations of functions such as `lookup`, `read` and `write` to access files, directories and links. The VFS allows file systems a great deal of flexibility in choosing which functions to implement and how to implement them.

20.7.3 Second Extended File System (ext2fs)

After its 1993 release, the **second extended file system (ext2fs)** quickly became the most widely used Linux file system of its time. The primary goal of ext2fs is to provide a high-performance, robust file system with support for advanced features.[113] As required by the virtual file system, ext2fs supports basic objects such as the superblock, inodes and directories. The ext2fs implementation of these objects extends their definitions to include specific information about the location and layout of data on disk, as well as providing functions to retrieve and modify data.

When an ext2fs partition is formatted, its corresponding disk space is divided into fixed-size blocks of data. Typical block sizes are 1,024, 2,048, 4,096 or 8,192 bytes. The file system stores all file data and metadata in these blocks.[114] By default, five percent of the blocks are reserved exclusively for users with root privileges when the disk is formatted. This is a safety mechanism provided to allow root processes to continue to run if a malicious or errant user process consumes all other available blocks in the file system.[115] The remaining 95 percent of the blocks can be used by all users to organize and store file data.

An **ext2 inode** represents files and directories in an ext2 file system—each inode stores information relevant to a single file or directory, such as time stamps, permissions, the identity of the file's owner and pointers to data blocks (Fig. 20.14). A single block is rarely large enough to contain an entire file. Thus, there are 15 data block pointers (each 32 bits wide) in each ext2 inode. The first 12 pointers directly locate the first 12 data blocks. The 13th pointer is an **indirect pointer**. The indirect pointer locates a block that contains pointers to data blocks. The 14th pointer is a **doubly indirect pointer**. The doubly indirect pointer locates a block of indirect pointers. The 15th pointer is a **triply indirect pointer**—a pointer to a block of doubly indirect pointers.

Consider an ext2 file system that uses 32-bit block addresses and a block size of 4,096 bytes. If a file is less than 48KB in size (i.e., it consumes 12 blocks of data or fewer), the file system can locate the file's data using pointers directly from the file's inode. The block of singly indirect pointers locates up to 1,024 data blocks (4MB of file data). Thus, the file system need load only two blocks (the inode and the block of indirect pointers) to locate over 4MB of file data. Similarly, the doubly indirect block of pointers locates up to $1{,}024^2$, or 1,048,576, data blocks (4GB of file data).

©1981–2004, Deitel & Associates, Inc. All rights reserved.

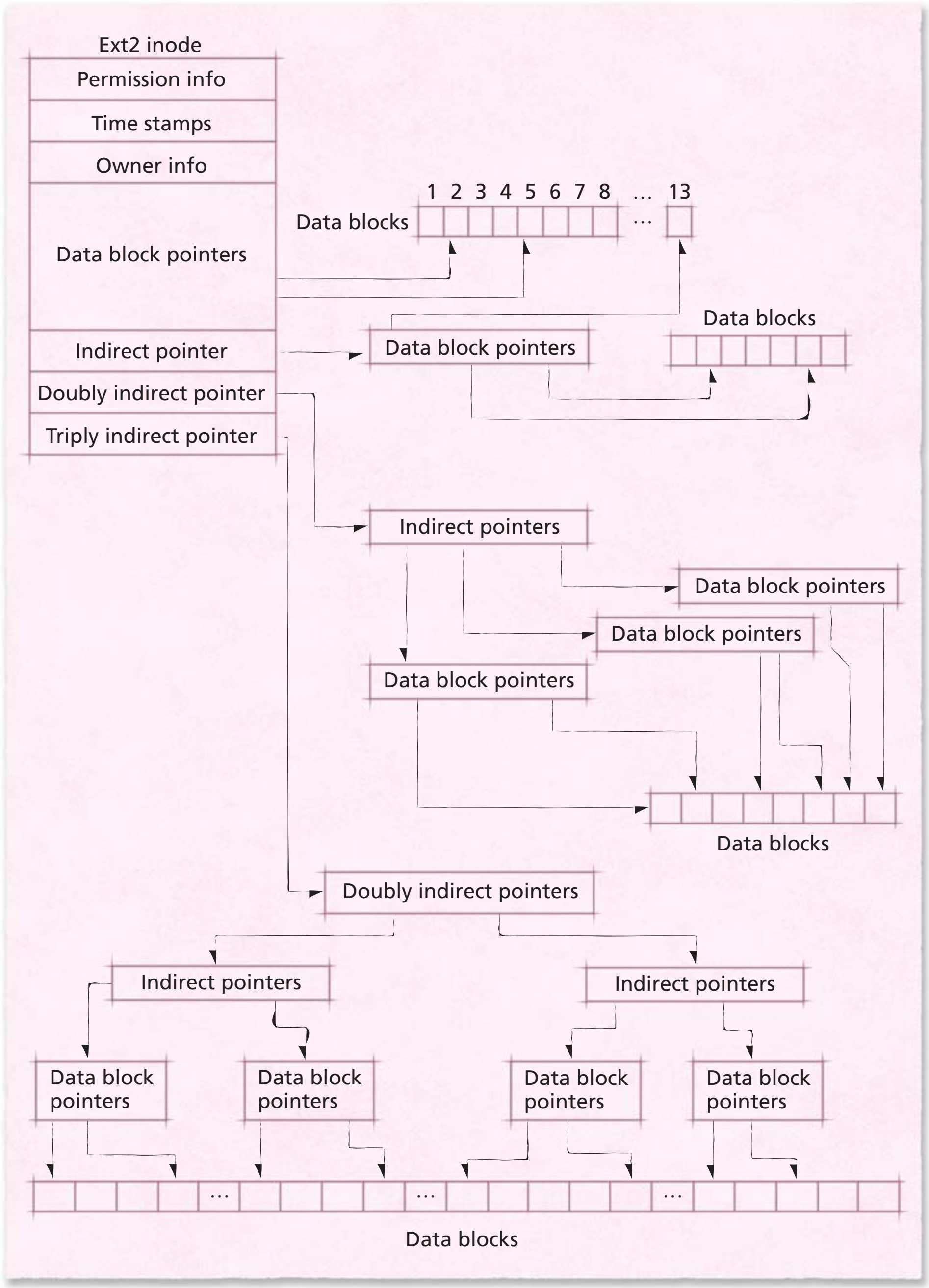


Figure 20.14 | *Ext2 inode contents.*

In this case, the file system must load one doubly indirect block, 1,025 singly indirect blocks and the file's inode (containing 12 direct pointers to data blocks) to access files of 4GB. Finally, the triply indirect block of pointers locates up to $1{,}024^3$,

©1981–2004, Deitel & Associates, Inc. All rights reserved.

or 1,073,741,824, data blocks (4,096GB of file data). In this case, the file system must load one triply indirect block, 1,025 doubly indirect blocks, 1,149,601 singly indirect blocks and the file's inode (containing 12 direct pointers to data blocks) to access files of approximately 4,100GB. This design provides fast access to small files, while supporting larger files (maximum file sizes range from 16GB to 4,096GB, depending on the file system's block size).[116]

Block Groups

Blocks in an ext2fs partition are divided into clusters of contiguous blocks called **block groups**. The file system attempts to store related data in the same block group. This arrangement reduces the seek time for accessing large groups of related data (e.g., directory inodes, file inodes and file data) because blocks inside each block group are located in a contiguous region of disk. Figure 20.15 illustrates the structure of a block group. The first block is the superblock. The superblock contains critical information about the entire file system, not just a particular block group. This information includes the total number of blocks and inodes in the file system, the size of the block groups, the time at which the file system was mounted and other housekeeping data. Because the contents of the superblock are critical to the integrity of the file system, a redundant copy of the superblock is maintained in some block groups. As a result, if any copy is corrupted, the file system can be restored from one of the redundant copies.[117]

The block group contains several data structures to facilitate file operations on that group. One such structure is the **inode table**, which contains an entry for each inode in the block group. When the file system is formatted, it assigns a fixed number of ext2 inodes to each block group. The number of inodes in the system depends on the ratio of bytes to inodes in the file system, specified when the partition is formatted. Because the size of the inode table is fixed, the only way to increase the number of inodes in a formatted ext2 file system is to increase the size of the file system. The inodes in each group's inode table typically point to file and directory data located in that group, reducing the time necessary to load files from disk due to the phenomenon of locality.

The block group also maintains a block containing an **inode allocation bitmap** that tracks inode use within a block group. Each bit in the allocation bitmap corresponds to an entry in the group's inode table. When a file is allocated, an available

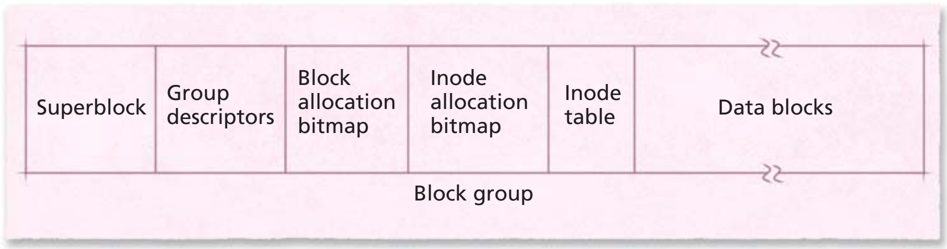


Figure 20.15 | Block group.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

inode is selected from the inode table to represent the file. The bit in the allocation bitmap corresponding to the inode's index in the inode table is turned on to indicate that the inode is in use. For example, if inode table entry 45 is assigned to a file, the 45th bit in the inode allocation bitmap is turned on. When an inode is no longer needed, the corresponding bit in the inode allocation bitmap is cleared to indicate that the inode can be reused. This same strategy is employed to maintain the **block allocation bitmaps**, which track each group's block usage.[118]

Another element of metadata in each block group, called the **group descriptor**, contains the block numbers corresponding to the location of the inode allocation bitmap, block allocation bitmap and inode table (Fig. 20.16). It also contains accounting information, such as the number of free blocks and inodes in the group. Each block group contains a redundant copy of its group descriptor for recovery purposes.[119]

The remaining blocks in each block group store file and directory data. Directories are variable-length objects that associate file names with inode numbers using **directory entries**. Each directory entry is composed of an inode number, directory entry length, file name length, file type and file name (Fig. 20.17). Typical file types include data files, directories and symbolic links; however, ext2fs also can use files to represent other objects, such as devices and sockets.[120]

Ext2fs supports both hard and symbolic links (recall from Section 13.4.2, Metadata, that symbolic links specify a pathname, not an inode number). When the

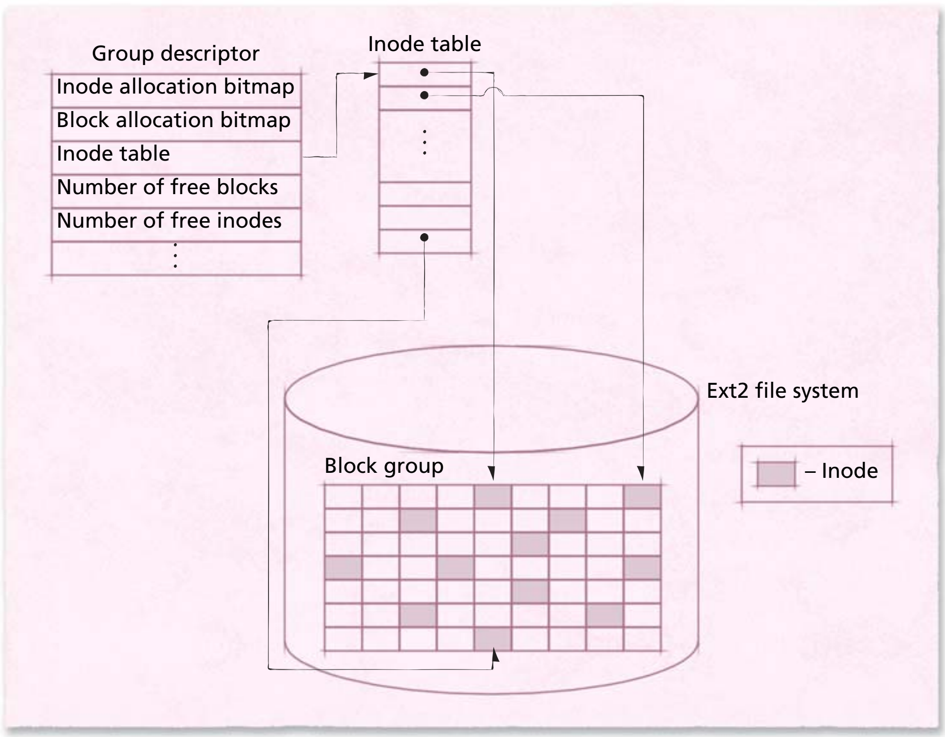


Figure 20.16 | *Group descriptor.*

©1981–2004, Deitel & Associates, Inc. All rights reserved.

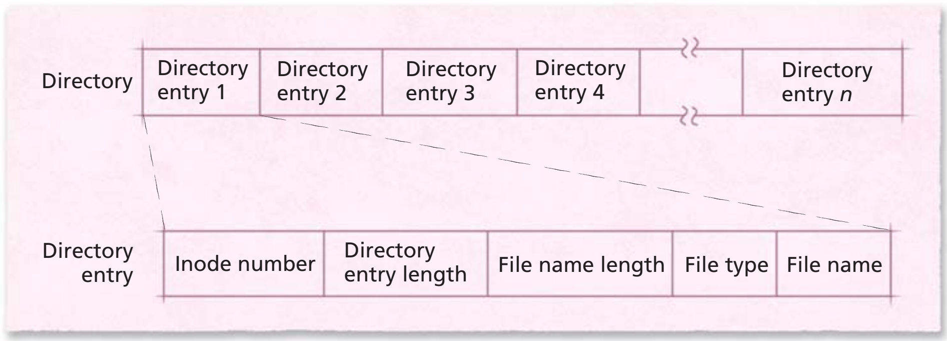


Figure 20.17 | Directory structure.

file system encounters a symbolic link while translating a pathname to an inode, the pathname being translated is replaced by the contents of the symbolic link, and the conversion is restarted. Because hard links specify an inode number, they do not require pathname-to-inode conversion. The file system maintains a count of the number of directory entries referencing an inode to ensure an inode is not deleted while it is still being referenced.[121]

File Security

Each file's inode stores information that the kernel uses to enforce its access control policies. In the ext2 file system, inodes contain two fields related to security: **file permissions** and **file attributes**. File permissions specify read, write and execute privileges for three categories of users: the owner of the file (initially the user that created the file), a group of users allowed to access the file (initially the group to which the user that created the file belongs), and all other users in the system.

File attributes control how file data can be modified. For example, the append-only file attribute specifies that users may append data to the file, but not modify data that already exists in the file. Ext2 file attributes can be extended to support other security features. For example, ext2 stores access control metadata in its extended file attributes to implement POSIX access control lists.[122]

Locating Data Using a Pathname

To locate a file in a file system, a pathname-to-inode conversion must be performed. Consider the example of finding the file given by the pathname `/home/admin/policydoc`. The pathname is composed of a series of directory names separated by slashes (`/home/admin`) that specify the path to the file `policydoc`. The conversion begins by locating the inode representing the root directory of the file system. The inode number of the root directory (`/`) is stored in the file system superblock (and is always 2).[123] This inode number specifies the root directory inode in the appropriate block group. The data blocks referenced by the latter inode contain the directory entries for the root directory. Next, the file system searches these directory entries for the inode number of the `home` directory. This

©1981–2004, Deitel & Associates, Inc. All rights reserved.

process is repeated until the inode representing the file `policydoc` is located. As each inode is accessed, the file system checks the permission information stored in it to ensure that the process performing the search is permitted to access the inode. The file's data can be directly accessed after the correct inode has been located and the file data (and metadata) have been cached by the system.[124]

20.7.4 Proc File System

One strength of the VFS is that it does not impose many restrictions on file system implementations. The VFS requires only that function calls to an underlying file system return valid data. This abstraction of a file system permits some intriguing file system implementations.

One such file system is **procfs (the proc file system)**. Procfs was created to provide real-time information about the status of the kernel and processes in a system. Similar to the VFS, the proc file system is created by the kernel at runtime.

The information provided by procfs can be found in the files and subdirectories within the `/proc` directory (Fig. 20.18). By examining the contents of `/proc`, users can obtain detailed information describing the system, from hardware status information to data describing network traffic.[125] For example, each number in Fig. 20.18 corresponds to a process in the system, identified by its process ID. By examining the contents of a process's directory in the proc file system, users can obtain information such as a process's memory usage or the location of its executable file. Other directories include `devices` (which contains information about devices in the system), `mounts` (which contains information regarding each mounted file system) and

```
root> ls /proc
1      20535  20656  751  978          interrupts  pci
10     20538  20657  792  acpi         iomem       self
137    20539  20658  8    asound       ioports     slabinfo
19902  20540  20696  811  buddyinfo    irq         stat
2      20572  20697  829  bus          kcore       swaps
20473  20576  20750  883  cmdline      kmsg        sys
20484  20577  3      9    cpuinfo      ksyms       sysvipc
20485  20578  4      919  crypto       loadavg     tty
20489  20579  469    940  devices      locks       uptime
20505  20581  5      960  dma          meminfo     version
20507  20583  536    961  dri          misc        vmstat
20522  20586  541    962  driver       modules
20525  20587  561    963  execdomains  mounts
20527  20591  589    964  filesystems  mtrr
20529  20621  6      965  fs           net
20534  20624  7      966  ide          partitions
root>
```

Figure 20.18 | Sample contents of the `/proc` directory.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

`uptime` (which displays the amount of time the system has been running). Data provided by procfs is particularly useful for driver developers and system administrators who require detailed information about system usage. In this section, we limit our discussion to the implementation of procfs. A detailed explanation of the `/proc` directory's contents can be found in the Linux source code under `Documentation/filesystems/proc.txt`.

Procfs is a file system that exists only in main memory. The contents of files in the proc file system are not stored persistently on any physical medium—procfs files provide users an access point to kernel information, which is generated on demand. When a file or directory is registered with the proc file system, a proc directory entry is created. **Proc directory entries**, unlike VFS directory entries, allow each directory to implement its own `read` function. This enables a proc directory to generate its contents each time the directory entry is accessed. When a process accesses a particular procfs file, the kernel calls the corresponding file operation specified by the file. These functions allow each file to respond differently to read and write operations.[126] The kernel creates many procfs entries by default. Additional files and directories can be created using loadable kernel modules.

When a user attempts to read data from a procfs file, the VFS calls the procfs `read` function, which accesses a proc directory entry. To complete a read request, procfs calls the `read` function defined for the requested file. Procfs `read` functions typically gather status information from a resource, such as the amount of time the system has been running. Once information has been retrieved by a `read` function, procfs passes the output to the process that requested it.[127]

Procfs files can be used to send data to the kernel. Some system variables, such as the network host name of a machine, can be modified at runtime by writing to procfs files. When a process writes to a procfs file, the data provided by the process may be used to update the appropriate kernel data structures.[128]

20.8 Input/Output Management

This section explains how the kernel accesses system devices using the I/O interface. The kernel abstracts the details of the hardware in a system, providing a common interface for I/O system calls. The kernel groups devices into classes; members of each device class perform similar functions. This allows the kernel to address the performance needs of certain devices (or classes of devices) individually.

20.8.1 Device Drivers

Support for devices such as graphics cards, printers, keyboards and other such hardware is a necessary part of any operating system. A device driver is the software interface between system calls and a hardware device. Independent Linux developers, not device manufacturers, have written most of the drivers that operate devices commonly found in Linux systems. This generally limits the number of devices that are compatible with the Linux operating system. As the popularity of Linux increases, so does the number of vendors that ship Linux drivers with their devices.

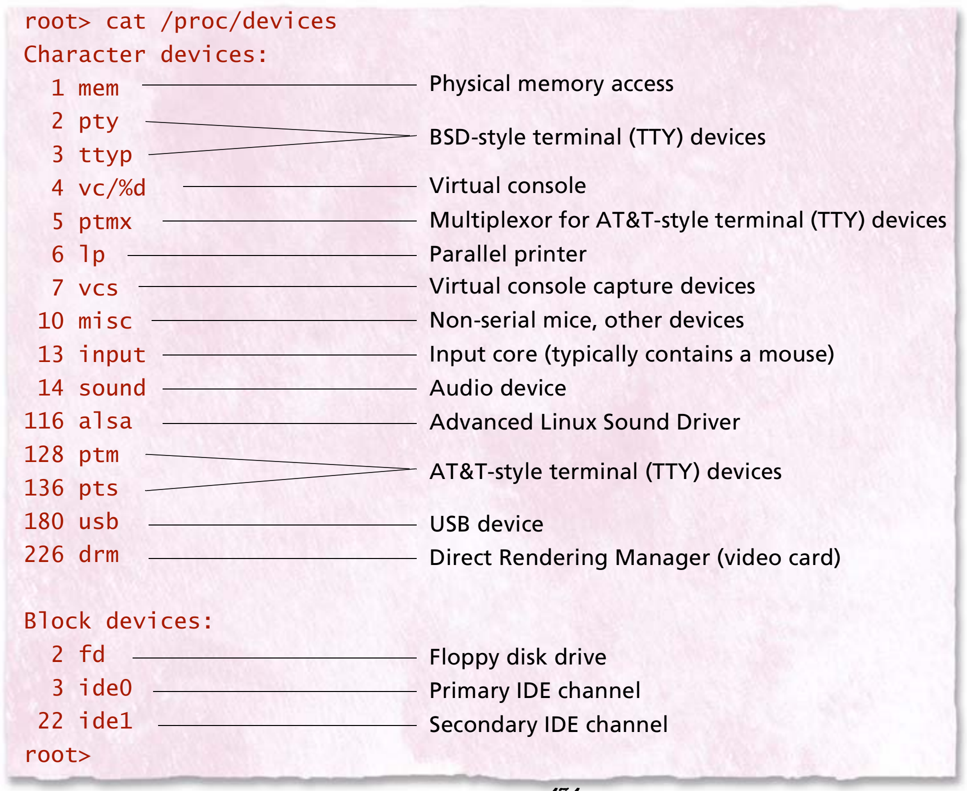
©1981–2004, Deitel & Associates, Inc. All rights reserved.

Typically, device drivers are implemented as loadable kernel modules. Drivers implemented as modules can be loaded and unloaded as they are needed, avoiding the need to have them permanently loaded in the kernel.[129]

Most devices in a system are represented by **device special files**. A device special file is an entry in the `/dev` directory that provides access to a particular device. Each file in the `/dev` directory corresponds to a block device or a character device.[130] A list of block and character device drivers that are currently loaded on a particular system can be found in the file `/proc/devices` (Fig. 20.19).

Devices that perform similar functions are grouped into **device classes**. For example, each brand of mouse that connects to the computer may belong to the input device class. To uniquely identify devices in the system, the kernel assigns each device a 32-bit device identification number. Device drivers identify their devices using a **major identification number** and a **minor identification number**. Major and minor identification numbers for all devices supported by Linux are located in the Linux Device List, which is publicly available online.[132] Driver developers must use the numbers allocated to devices in the Linux Device List to ensure that devices in a system are properly identified.

Devices that are assigned the same major identification number are controlled by the same driver. Minor identification numbers allow the system to distin-



```
root> cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 vc/%d
  5 ptmx
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
116 alsa
128 ptm
136 pts
180 usb
226 drm

Block devices:
  2 fd
  3 ide0
 22 ide1
root>
```

Physical memory access
BSD-style terminal (TTY) devices
Virtual console
Multiplexor for AT&T-style terminal (TTY) devices
Parallel printer
Virtual console capture devices
Non-serial mice, other devices
Input core (typically contains a mouse)
Audio device
Advanced Linux Sound Driver
AT&T-style terminal (TTY) devices
USB device
Direct Rendering Manager (video card)
Floppy disk drive
Primary IDE channel
Secondary IDE channel

Figure 20.19 | `/proc/devices` file contents.[131]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

guish individual devices that are assigned the same major identification number (i.e., belong to the same device class).[133] For example, a hard disk is assigned a major identification number, and each partition on the hard disk is assigned a device minor number.

Device special files are accessed via the virtual file system. System calls pass to the VFS, which in turn issues calls to device drivers. Figure 20.20 illustrates the interaction between system calls, the VFS, device drivers and devices. Drivers implement generic virtual file system functions so that the processes may access `/dev` files using standard library calls. For example, standard library calls for accessing files (such as printing to a file using the standard C library function `fprintf`) are implemented on top of lower-level system calls (such as `read` and `write`).[134] Individual device characteristics determine the drivers and their corresponding system calls necessary to support the device. Most devices that Linux supports belong to three primary categories: character devices, block devices and network devices.

Each class of device requires its corresponding device drivers to implement a set of functions common to the class. For example, a character device driver must implement a `write` function to transfer data to its device.[135] Further, if a device is attached to a kernel subsystem (e.g., the SCSI subsystem), the device's drivers must interface with the subsystem to control the device. For example, a SCSI device driver passes I/O requests to the SCSI subsystem, which then interacts directly with devices attached to the SCSI interface.[136] Subsystem interfaces exist to reduce redundant code; for example, each SCSI device driver need only provide access to a particular

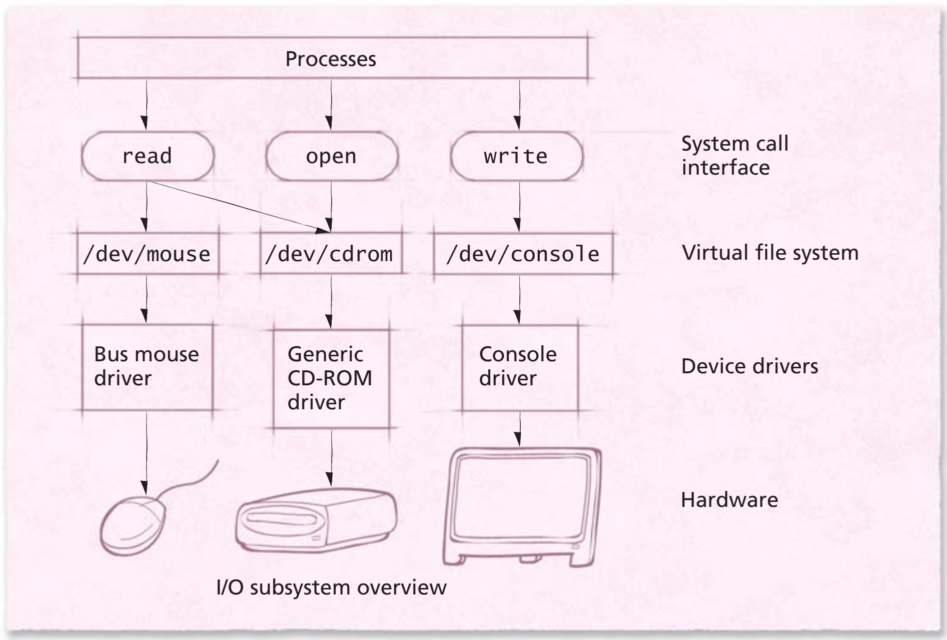


Figure 20.20 | I/O interface layers.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

SCSI device, not to the SCSI controller to which it is attached. The SCSI subsystem provides access to common SCSI components, such as the SCSI controller.

Most drivers implement common file operations such as `read`, `write` and `seek`. These operations allow drivers to transfer data to and from devices, but do not allow them to issue hardware-specific commands. To support tasks such as ejecting a CD-ROM tray or retrieving status information from a printer, Linux provides the `ioctl` system call. `ioctl` allows developers to send control messages to any device in the `/dev` directory. The kernel defines several default control messages and allows a driver to implement its own messages for hardware-specific operations.[137] The set of messages supported by a device driver is dependent on the driver's device and implementation.

20.8.2 Character Device I/O

A **character device** transmits data as a stream of bytes. Devices that fall under this category include printers, consoles, mice, keyboards and modems. Because they transfer data as streams of bytes, most character devices support only sequential access to data.[138]

Most character device drivers implement basic operations such as opening, closing, reading from and writing to a character device. Each device in the system is represented by a **`device_struct`** structure that contains the driver name and a pointer to the driver's `file_operations` structure, which maintains the operations supported by the device driver. To initialize a character device, a device driver must register its operations with the virtual file system, which appends a `device_struct` structure to the array of registered drivers stored in **`chrdevs`**.[139] Figure 20.21

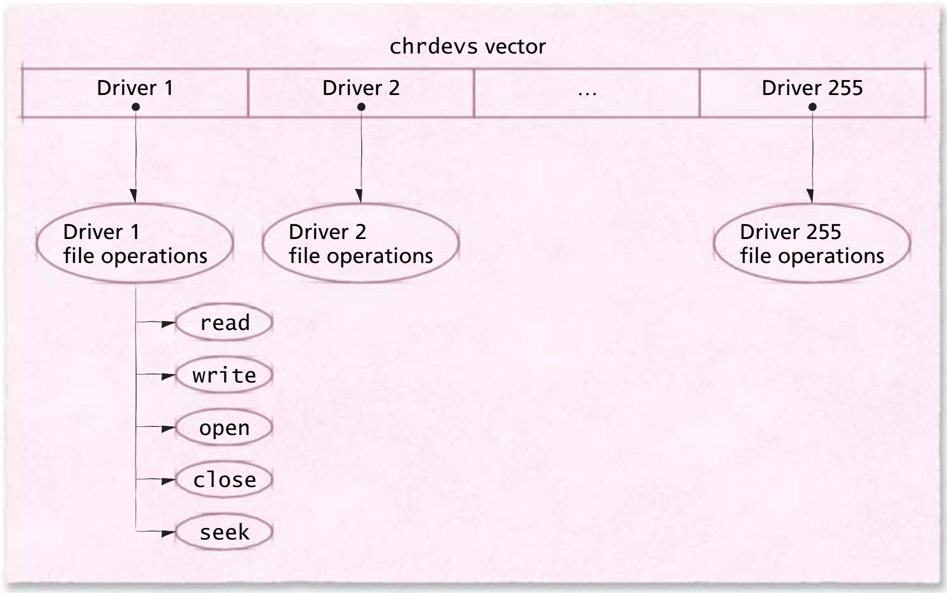


Figure 20.21 | chrdevs vector.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

describes the contents of vector `chrdevs`. Each entry in `chrdevs` corresponds to a device driver major identification number. For example, the fifth entry in `chrdevs` is the `device_struct` for the driver with major number five.[140]

When a system call accesses a device special file, the VFS calls the appropriate function in the device's `file_operations` structure. This structure includes functions that perform read, write and other operations on the device. The inode representing the file stores a device special file's `file_operations` structure.[141]

After a device's file operations have been loaded into its inode, the VFS will use those operations whenever system calls access the device. The system can access this inode until a system call closes the device special file. However, once a system call closes the file, the inode must be recreated and initialized the next time the file is opened.[142]

20.8.3 Block Device I/O

Unlike character devices, block devices allow data stored in fixed-sized blocks of bytes to be accessed at any time, regardless of where those blocks are stored on the device. To facilitate nonsequential (i.e., random) access to a large amount of data (e.g., a file on a hard drive), the kernel must employ a more sophisticated system for handling block device I/O than it does for handling character device I/O. For example, the kernel provides algorithms that attempt to optimize moving-head storage (i.e., hard disks).

Like character devices, block devices are identified by major and minor numbers. The kernel's block I/O subsystem contains a number of layers to modularize block I/O operations by placing common code in each layer. Figure 20.22 depicts the layers through which block I/O requests pass. To minimize the amount of time spent accessing block devices, the kernel uses two primary strategies: caching data and clustering I/O operations.

Buffering and Caching

To reduce the number of block I/O operations for disk devices, the kernel buffers and caches I/O requests. When a process requests data from a block device (typi-

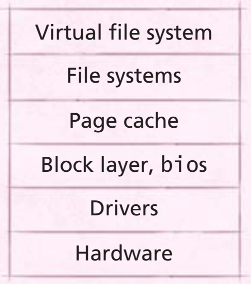


Figure 20.22 | Block I/O subsystem layers.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

cally a hard disk), the kernel searches the page cache for the requested blocks. (Recall from Section 20.6.3, Page Replacement, that the page cache is a region of main memory that stores buffered and cached data from I/O requests.) If the page cache contains an entry for the requested block, the kernel will copy that page to the user's virtual space, provided there are no errors (e.g., improper permission). When a process attempts to write data to a block device, the request is typically placed on a list of pending requests that is sorted according to the kernel's disk scheduling strategy.

By performing **direct I/O**, a driver can bypass kernel caches when reading from (or writing to) devices. Some applications, such as high-end database applications, implement their own caching mechanisms; as a result, it would be wasteful for the kernel to maintain its own cache of the application's data.[158] When direct I/O is enabled, the kernel performs I/O directly between a process's user address space and the device, eliminating the overhead caused by copying data from a user address space to the kernel caches, then to the device.

Request Lists

If an I/O request corresponds to data that is not cached or data that must be written to secondary storage, the kernel must perform an I/O operation. Instead of submitting I/O requests to devices in the order in which they are received, the kernel adds a request to a **request list**. A request list, which contains pending I/O operations, is created for each block device in the system. The list allows the kernel to order requests to take into account factors such as the location of the disk head if the block device is a hard disk. As discussed in Chapter 12, the kernel can improve the performance of all block I/O operations by sorting requests for I/O on each block device.

To associate entries in the request list with page frames, each request contains a **`bio` structure**, which maps to a number of pages in memory corresponding to the request. The kernel maintains at least one request list per driver; each request corresponds to a read or write operation.[143, 144] Block drivers do not define read and write operations, but rather must implement a request function that the kernel calls, once it has queued requests.[145] This allows the kernel to improve I/O performance by sorting the list of requests according to its disk scheduling algorithm (discussed in the next section) before submitting requests to a block device. When the kernel calls a request function, the block device must perform all I/O operations in the list of I/O requests the kernel provides.

Although the kernel often reduces seek time by sorting block device requests, in some cases the request list is detrimental to performance. For example, certain device drivers, such as RAID drivers, implement their own methods for managing requests (see Section 12.10, Redundant Arrays of Independent Disks (RAID)). Such device drivers operate on `bio`s, unlike traditional block device drivers (e.g., IDE), which are passed a list of requests via a request function.[157]

Elevator Disk Scheduling Algorithm

Linux provides several disk scheduling algorithms to allow users to customize I/O performance to meet the individual needs of each system. The default disk schedul-

©1981–2004, Deitel & Associates, Inc. All rights reserved.

ing algorithm is a variation of the elevator algorithm (i.e., the LOOK variation of the SCAN strategy presented in Section 12.5.6, LOOK and C-LOOK Disk Scheduling). To minimize disk seek time, the kernel arranges the entries in the list according to their location on disk. The request at the head of the list is closest to the disk head, which reduces the amount of time the disk spends seeking and increases I/O throughput. When an I/O request is submitted to a disk's request list, the kernel determines the location on disk corresponding to the request. The kernel then attempts to **merge** requests to adjacent locations on disk by combining two I/O requests into a single, larger request. Merging requests improves performance by reducing the number of I/O requests issued to a block device. If a request cannot be merged, the kernel attempts to insert that request in the sorted list in the position that maintains the list's least-seek-time-first ordering.[146]

Although the elevator algorithm provides high throughput by reducing disk seek latency, the algorithm allows requests at the end of the queue to be indefinitely postponed.[147] For example, consider two processes: process P_1 writes 200MB of data to a file and process P_2 recursively reads the contents of a directory on disk and prints the result to the terminal. Assume that the system is using an ext2 file system and the request list is initially empty. As P_1 executes, it may submit several write requests without blocking during its time slice—processes rarely block as result of write requests because they do not rely on the completion of write operations to execute subsequent instructions. P_1 is eventually preempted, at which point the request list contains several write requests. Many of the write requests will have been merged by the kernel because the ext2 file system attempts to locate file data within block groups, as discussed in Section 20.7.3, Second Extended File System (ext2fs). Process P_1's requests are then submitted to the block device, which moves the disk head to the location of the data blocks to be written.

When P_2 executes, it submits a request to read the contents of a directory. This request is a synchronous read operation because P_2 cannot print the directory contents until the read operation completes. Consequently, P_2 will submit a single I/O request and block. Unless the read request corresponds to a location adjacent to the disk head (which is now servicing P_1's write requests), the read request is placed after the pending write requests in the request list. Because P_2 has blocked, process P_1 may eventually regain control of the processor and submit additional write requests. Each subsequent write request will likely be merged with the previous requests at the front of the request list. As a result, process P_2's read request is pushed further back in the request list while P_2 remains blocked—meaning that P_2 cannot submit additional I/O requests. As long as process P_1 continues to submit write requests, process P_2's read request is indefinitely postponed.

Deadline and Anticipatory Disk Scheduling Algorithms

To eliminate indefinite postponement, the kernel provides two LOOK disk scheduling algorithms: deadline scheduling and anticipatory scheduling. The **deadline scheduler** prevents read requests from being indefinitely postponed by assigning each request a deadline—the scheduler attempts to service each request before its

©1981–2004, Deitel & Associates, Inc. All rights reserved.

deadline passes. When the request has been waiting for the longest time permitted by the deadline, the request **expires**. The deadline scheduler processes requests from the head of the request list unless a request expires. At this point, the deadline scheduler services any requests that have expired and some that will soon expire. Servicing requests that will soon expire reduces the number of times the scheduler will be interrupted from servicing requests at the head of the list, which improves throughput. After all expired requests have been serviced, the scheduler continues by servicing requests from the front of the request list.[148]

To meet its deadlines, the scheduler must be able to quickly determine if any requests have expired. Using a single request list, the deadline scheduler would need to perform a linear search of requests to determine if any had expired. If the number of requests in the request list is large, this could require a significant amount of time, leading to poor performance and missed deadlines. Consequently, the deadline scheduler maintains two FIFO queues, one each for read and write requests. When a request is added to the request list, a reference to the request is added to the appropriate FIFO queue. Therefore, the request that has waited for the longest time is always located at the front of the FIFO queue. This means that the deadline I/O scheduler can quickly determine if a request is near its deadline by accessing the pointer to the front of each of the FIFO queues.[149, 150]

Because the deadline scheduler is designed to prevent read starvation, the deadline for read requests is shorter than the deadline for write requests. By default, read requests must be serviced 500ms after insertion into the request list, whereas write requests must be serviced after 5 seconds. These values were chosen because they provide good performance in general, but they can be modified by a system administrator at run time.[151]

Consider how the deadline scheduler performs given processes P_1 and P_2 from the previous section (process P_1 writes 200MB of data to a file and process P_2 recursively reads the contents of a directory on disk and prints the result to the screen). Recall from the previous section that process P_1's write requests are typically performed before read requests because its requests are merged. As a result, the disk will most likely be servicing a write request when P_2's read request's deadline expires. This means that the disk will likely perform a seek operation to perform the read request. Also, because synchronous read requests require process P_2 to block, the number of read requests in the request list is small. Consequently, the next request in the request list following the read operation will likely be a write operation that requires another seek operation. If several processes issue read requests while one or more processes submit write requests, several such pairs of seek operations (due to expired read requests) may occur within a short period of time. Thus, to reduce the number of seeks, the deadline I/O scheduler attempts to group several expired requests so that they will be serviced together before expired write requests are serviced, and vice versa.[152, 153]

The anticipatory scheduler eliminates read request starvation by preventing excessive seek activity and further improves performance by anticipating future

©1981–2004, Deitel & Associates, Inc. All rights reserved.

requests. Recall that synchronous read requests often occur once per time slice because they require processes to block. However, similar to process P_2's requests for directory entries, many processes issue a series of synchronous read requests for contiguous data (or data on a single track). Consequently, if the disk scheduler paused briefly after completing a process's read request, that process may issue an additional read request that does not require a seek operation.[154] Even when there are several other requests in the request list, this read request would be placed at the head of request list and serviced without causing excessive seek activity.

By default, the amount of time during which the anticipatory I/O scheduler waits for a new request is 6ms—a pause that occurs only after completing a read request.[155] The 6ms pause (the value of the pause can be modified at run time) corresponds to the seek latency for many of today's hard disks, or roughly half the amount of time required to perform a request located on another track and return to the location of the previous read. If a read request is issued during the 6ms pause, the anticipatory scheduler can perform the request before seeking to another location on disk to perform requests for other processes. In this case, a traditional elevator scheduler would have performed two seek operations: one to perform the next request from another process and one to service the next read request. Therefore, the anticipatory I/O scheduler improves overall I/O throughput if it receives a read request within the 6ms pause more than 50 percent of the time.

The anticipatory scheduler has been shown to perform 5 to 100 times better than the traditional elevator algorithm when performing synchronous reads in the presence of write requests. However, the anticipatory I/O scheduler can introduce significant overhead due to its 6ms pause, leading to reduced I/O throughput. This occurs when the I/O scheduler does not receive a request for data near the disk's read/write head during the 6ms that it waits for a read request. To minimize this overhead, the scheduler maintains a history of process behavior that it uses to predict whether a process will benefit from the 6ms pause.[156]

20.8.4 Network Device I/O

The kernel's networking subsystem provides an interface for exchanging data with other hosts. This interface, however, cannot be accessed directly by user processes, which must send and receive data via the IPC subsystem's socket interface (discussed in Section 20.10.3, Sockets). When processes submit network data to the socket interface, they specify the network address of the destination, not the network device through which to deliver the data. The networking subsystem then determines which network device will deliver the packet. An important difference between network devices and block or character devices is that the kernel does not request data from a network device. Instead, network devices use interrupts to notify the kernel as they receive packets.

Because network traffic travels in packets, which can arrive at any time, the read and write operations of a device special file are not sufficient to access data from network devices. Instead, the kernel uses `net_device` structures to describe

©1981–2004, Deitel & Associates, Inc. All rights reserved.

the network devices.[159] These objects are similar to the `device_struct` objects that represent block and character devices; however, because network devices are not represented as files, the `net_device` structure does not include a `file_operations` structure. Instead, it contains a number of functions defined by drivers that allow the kernel to perform actions such as starting a device, stopping a device and sending packets to a device.[160]

Once the kernel has prepared packets to transmit to another host, it passes them to the device driver for the appropriate network interface card (NIC). To determine which NIC will send the packet, the kernel examines an internal routing table that lists the destination addresses each network interface card can access. Once the kernel matches the packet's destination address to the appropriate interface in the routing table, the kernel passes the packet to the device driver. Each driver processes packets according to a queuing discipline, which specifies the order in which its device processes packets, such as the default FIFO policy, or other, more sophisticated, priority-based policies. By enabling priority queuing disciplines, the system can deliver higher-priority content, such as streaming media, more quickly than other network traffic.[161]

After passing packets to a network device's queue, the kernel wakes the device so that the driver may begin removing packets from the device's queue according to its queuing discipline. As packets are removed from the queue, they are passed to a packet transmission function specified by the device's driver.[162]

When a network interface receives a packet from an external source, it issues an interrupt. The interrupt causes processor control to pass to the appropriate interrupt handler for packet processing. The interrupt handler allocates memory for the packet, then passes the packet to the kernel's networking subsystem. In Section 20.11, Networking, we discuss the path taken by packets as they travel through the networking subsystem.

20.8.5 Unified Device Model

The **unified device model** is an attempt to simplify device management in the kernel. At the physical level, devices are attached to an interface (e.g., a PCI slot or a USB port) that is connected to the rest of the system via a bus. As discussed in the previous sections, Linux represents devices as members of device classes. For example, a mouse connected to a USB port and a keyboard connected to a PS/2 port both belong to the input device class, but each device connects to the computer via a different bus. Whereas a description of device interfaces and buses is a physical view of the system, a device class is a software (i.e., abstract) view of the system that simplifies device management by grouping devices of a similar type.

Before the unified device model, device classes were not related to system buses, meaning that it was difficult for the system to determine where in the system a device was physically located. This was not a problem when computers did not support **hot swappable devices** (i.e., devices that can be added and removed while the computer is running). In the absence of hot swappable devices, it is sufficient for the

©1981–2004, Deitel & Associates, Inc. All rights reserved.

system to detect devices exactly once (at boot time). Once the kernel loads a driver corresponding to each device, the kernel rarely needs to access a device's physical location in the system. However, in the presence of hot swappable devices, the kernel must be aware of the physical layout of the system so it can detect when a device has been added or removed. Once it has located a new device, the kernel must be able to identify its class so the kernel can load the appropriate device driver.[163]

For example, devices are commonly added and removed from the USB bus. To detect such changes, the kernel must periodically poll the USB interface to determine which devices are attached to it.[164] If a new device is found, the kernel should identify it and load a device driver that supports it so that processes may use the device. If a device has been removed, the system should unregister the driver so that attempts to access the device are denied. Thus, the kernel must maintain a layout of the physical location of devices in the system so that it knows when the set of devices in the system changes. To support hot swappable devices, the kernel uses the unified device model to access the physical location of a device in addition to its device class representation.[165]

The unified device model defines data structures to represent devices, device drivers, buses, and device classes. The relationship between these structures is shown in Fig. 20.23. Each bus data structure represents a particular bus (e.g., PCI) and contains pointers to a list of devices attached to the bus and drivers that operate devices on the bus. Each class data structure represents a device class and contains pointers to a list of devices and device drivers that belong to that class. The unified device model associates each device and device driver with a bus and class. As a result, the kernel can access a bus, determine a list of devices and drivers on that bus and then determine the class to which each device and device driver belongs. Similarly, the kernel can access a device class, follow the pointers to its list of devices and device drivers and determine the bus to which each device is attached. As Fig. 20.23 demonstrates, the kernel requires a reference only to a single data structure to access all other data structures in the unified device model. This simplifies device management for the kernel as devices are added to and removed from the system.

When devices are registered with the system, these data structures are initialized and corresponding entries are placed in the **system file system, sysfs**. Sysfs provides an interface to devices described by the unified device model. Sysfs allows user applications to view the relationship between entities (devices, device drivers, buses and classes) in the unified device model.[166, 167, 168, 169]

Sysfs, typically mounted at `/sys`, organizes devices according to both the bus to which they are attached and the class to which they belong. The `/sys/bus` directory contains entries for each bus in the system (e.g., `/sys/bus/pci`). Within each bus subdirectory is a list of devices and device drivers that use the bus. Sysfs also organizes devices by class in the directory `/sys/class`. For example, the `/sys/class/input` contains input devices, such as a mouse or keyboard. Within each class subdirectory is a list of devices and device drivers that belong to that class.[170, 171]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

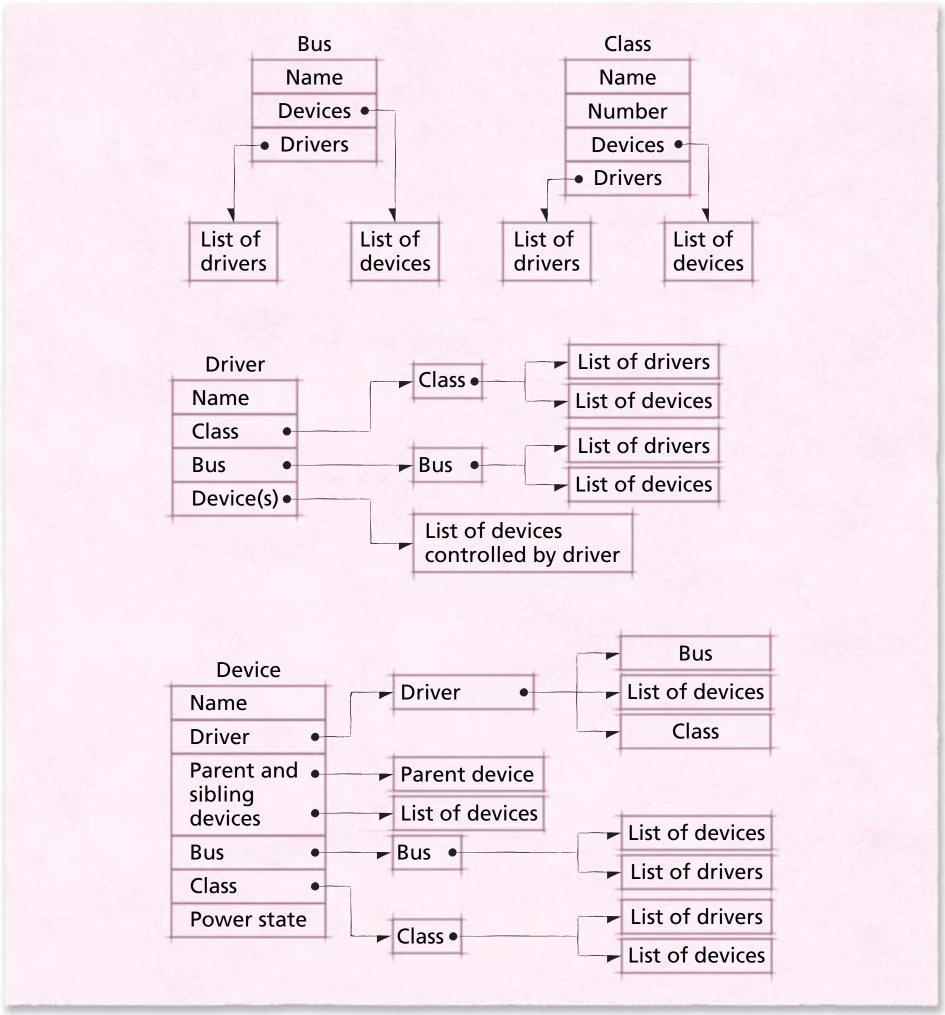


Figure 20.23 | Unified device model organization.

Power Management

The unified device model has also simplified how the kernel performs power management, an important consideration for battery-powered systems. As the number and power of mobile computers increases, a system's ability to manage power to increase battery life has become more important. Power management standards such as the Advanced Configuration and Power Interface (ACPI) specify several device power states, each of which results in different power consumption. For example, the ACPI defines four power states from fully on (D0) to fully off (D3). When a device transitions from fully on to fully off, the system does not provide any power to the device and any volatile data stored in the device, known as the device context, is lost.[172] The unified device model simplifies power management for the

©1981–2004, Deitel & Associates, Inc. All rights reserved.

kernel by providing a data structure to store the context for each device when its power state changes.

There are several devices that provide powered connections to other devices. For example, a PCI card that contains USB ports may contain connections to USB devices. Because some USB devices are powered through the USB cable, if power is removed from the PCI card, then power is removed from each of its attached devices. Consequently, the PCI card should not enter the fully off state until each of its attached devices has entered the fully off state. The unified device model allows the kernel to detect such power dependencies by exposing the physical structure of devices in the system. As a result, the kernel can prevent a device from transitioning to a different power state if the transition will prevent other devices from properly functioning.[173]

20.8.6 Interrupts

The kernel requires that each device driver register interrupt handlers when the driver is loaded. As a result, when the kernel receives an interrupt from a particular device, the kernel passes control to its corresponding interrupt handler. Interrupt handlers do not belong to any single process context because they are not themselves programs. Because an interrupt handler is not identified as any `task_struct` object's executable code, the scheduler cannot place it in any run queues. This characteristic places some restrictions on interrupt handlers; lacking its own execution context, an interrupt handler cannot sleep or call the scheduler. If an interrupt handler were permitted to sleep or call the scheduler, it never would regain control of the processor.

Similarly, interrupt handlers cannot be preempted, as doing so would invoke the scheduler.[174] Any preemption requests received during interrupt handling are honored when the interrupt handler completes execution. Finally, interrupt handlers cannot cause exceptions or faults while executing. In many architectures, the system aborts when an exception is raised during an interrupt handler.[175]

To improve kernel performance, most drivers attempt to minimize the processor cycles required to handle hardware interrupts. This is another reason why kernel memory is never swapped to disk—loading a nonresident page while an interrupt handler is executing takes substantial time. In Linux, a driver handling one interrupt cannot be preempted by other interrupts that use the same interrupt line. If this were not the case, any device driver containing nonreentrant code might perform operations that leave a device in an inconsistent state. As a result, when one driver is processing an interrupt, the kernel queues or drops any other interrupts it receives that use the same interrupt line.[176] Therefore, driver developers are encouraged to write code that processes interrupts as quickly as possible.

The kernel helps improve interrupt-handling efficiency by dividing interrupt-handling routines into two parts—the **top half** and the **bottom half**. When the kernel receives an interrupt from a hardware device, it passes control to the top half of the driver's interrupt handler. The top half of an interrupt handler performs the mini-

©1981–2004, Deitel & Associates, Inc. All rights reserved.

mum work required to acknowledge the interrupt. Other work (such as manipulating data structures)—which should be located in the bottom half of the interrupt handler—is scheduled to be performed later by a **software interrupt handler**. Top halves of interrupt routines cannot be interrupted by software interrupt handlers.

Two primary software interrupt-handler types are **softirqs** and **tasklets**. Softirqs can be executed concurrently on multiple processors (up to one per processor), making them ideal for SMP systems.[177] When a device driver allocates a softirq, it specifies the action to be performed each time the softirq is scheduled. Because multiple copies of a softirq can run simultaneously, softirq actions must be reentrant to perform reliably. Network devices on Web servers, which constantly receive packets of data from external sources, benefit from softirqs because multiple packets can be processed simultaneously on different processors.[178]

However, softirqs do not improve performance for several types of interrupt handlers. For example, a driver that requires exclusive access to data would need to enforce mutual exclusion if its code were executed simultaneously on multiple processors. In some cases, the overhead due to enforcing mutual exclusion can outweigh the benefits of multiprocessing. Other devices transfer data as a series of bits, requiring device drivers to sequentialize access to such data. Such devices cannot benefit from parallel processing and consequently use tasklets to perform bottom-half interrupt-handling routines.[179]

Tasklets are similar to softirqs but cannot run simultaneously on multiple processors and therefore cannot take advantage of parallel processing. As a result, most drivers use tasklets instead of softirqs to schedule bottom halves. Although multiple instances of a single tasklet cannot execute simultaneously, several different tasklets can execute simultaneously in SMP systems.[180]

Softirqs and tasklets normally are handled in interrupt context or in a process's context immediately after the top-half interrupt handler completes, executing with higher priority than user processes. If the system experiences a large number of softirqs that reschedule themselves, user processes might be indefinitely postponed. Thus, when user processes have not executed for a significant period of time, the kernel assigns softirqs and tasklets to be executed by the kernel thread ***ksoftirqd***, which executes with low-priority (+19). When the kernel is loaded, the Linux kernel creates an instance of the kernel thread *ksoftirqd* for each processor. These threads remain sleeping until the kernel wakes them. Once scheduled, *ksoftirqd* enters a loop in which it processes pending tasklets and softirqs sequentially. *ksoftirqd* continues processing tasklets and softirqs until the tasklets and/or softirqs have completed execution or until *ksoftirqd* is preempted by the scheduler.[181]

Partitioning interrupt handling into top and bottom halves minimizes the amount of time that hardware interrupts are disabled. Once a driver handles a hardware interrupt (the top half), the kernel can run the software interrupt handler (the bottom half), during which incoming interrupts can preempt the software interrupt handler.[182] This division of driver code improves the response time of a system by reducing the amount of time during which interrupts are disabled.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.9 Kernel Synchronization

A process executing in user mode cannot directly access kernel data, hardware, or other critical system resources—such processes must rely on the kernel to execute privileged instructions on their behalf. These operations are called **kernel control paths**. If two kernel control paths were to access the same data concurrently, a race condition could result.[183] To prevent this, the kernel provides two basic mechanisms for providing mutually exclusive access to critical sections: locks and semaphores.

20.9.1 Spin Locks

Spin locks allow the kernel to protect critical sections in kernel code executing on SMP-enabled systems. Before entering its critical section, a kernel control path acquires a spin lock. The region remains protected by the spin lock until the kernel control path releases the spin lock. If a second kernel control path attempts to acquire the same spin lock to enter its critical section, it will enter a loop in which it busy waits, or "spins," until the first kernel control path releases the spin lock. Once the spin lock becomes available, the second kernel control path can acquire it.[184]

Proper use of spin locks prevents race conditions among multiple kernel control paths executing concurrently in an SMP system, but serves no purpose in a uniprocessor system in which two kernel control paths cannot simultaneously execute. Consequently, kernels configured for uniprocessor systems exclude the locking portion of spin lock calls.[185] This improves performance by eliminating the costly instructions executed to acquire mutually exclusive access to a critical section in multiprocessor systems.

The kernel provides a set of spin lock functions for use in interrupt handlers. Because a hardware interrupt can preempt any execution context, any data shared between a hardware interrupt handler and a software interrupt handler must be protected using a spin lock. To address this issue, the kernel provides spin locks that disable interrupts on the local processor while still allowing concurrent execution on SMP systems. On uniprocessor systems, the spin lock code is removed when the kernel is compiled, but the code for enabling and disabling interrupts remains intact.[186] To protect data shared between user contexts and software interrupt handlers, the kernel uses **bottom-half spin locks**. These functions disable software interrupt handlers in addition to acquiring the requested spin lock.[187]

All spin lock variations disable preemption in both single and multiprocessor systems. Although disabling preemption could lead to indefinite postponement or even deadlock, allowing code protected by spin locks to be preempted introduces the same race conditions spin locks are designed to avoid. The kernel uses a preemption lock counter to determine if a kernel control path can be preempted. When a kernel control path acquires a spin lock, the preemption lock counter is incremented; the counter is decremented when the kernel control path releases the spin lock. Code executing in kernel mode can be preempted only when the preemption lock counter is reduced to zero. When a spin lock is released and its associated

©1981–2004, Deitel & Associates, Inc. All rights reserved.

counter becomes zero, the kernel honors any pending preemption requests by invoking the scheduler.

Kernel developers must abide by certain rules to avoid deadlock when using any spin lock variant. First, if a kernel control path has already acquired a spin lock, the kernel control path must not attempt to acquire the spin lock again before releasing it. Attempting to acquire the spin lock a second time will cause the kernel control path to busy wait for the lock it controls to be released, causing deadlock. Similarly, a kernel control path must not sleep while holding a spin lock. If the next task that is scheduled attempts to acquire the spin lock, deadlock will occur.[188]

20.9.2 Reader/Writer Locks

In some cases, multiple kernel control paths need only to read (not write) the data accessed inside a critical section. When no kernel control path is modifying that data, there is no need to prevent concurrent read access to the data (see Section 6.2.4, Monitor Example: Readers and Writers). To optimize concurrency in such a situation, the kernel provides **reader/writer locks**. Reader/writer spin locks and kernel semaphores (Section 20.9.4, Kernel Semaphores) allow multiple kernel control paths to hold a read lock, but permit only one kernel control path to hold a write lock with no concurrent readers. A kernel control path that holds a read lock on a critical section must release its read lock and acquire a write lock if it wishes to modify data.[189] An attempt to acquire a write lock succeeds only if there are no other readers or writers concurrently executing inside their critical sections. Reader/writer locks effect a higher level of concurrency by limiting access to a critical section only when writes occur. Depending on the kernel control paths accessing the lock, this can lead to improved performance. If readers do not release their read locks, it is possible for writers to be indefinitely postponed. To prevent indefinite postponement and provide writers with fast access to critical sections, kernel control paths use the seqlock.

20.9.3 Seqlocks

In some situations, the kernel employs another locking mechanism designed to allow writers to access data immediately without waiting for readers to release the lock. This locking primitive, called a **seqlock**, represents the combination of a spin lock and a sequence counter. Writing to data protected by a seqlock is initiated by calling function `write_seqlock`. This function acquires the spin lock component of the seqlock (so that no other writers can enter their critical section) and it increments the sequence counter. After writing, function `write_sequnlock` is called, which once again increments the sequence counter, then releases the spin lock.[190]

To enable writers to access data protected by a seqlock immediately, the kernel does not allow readers to acquire mutually exclusive access to that data. Thus, a reader executing its critical section can be preempted, enabling a writer to modify the data protected by a seqlock. The reader can detect if a writer has modified the value of the data protected by the seqlock by examining the value of the seqlock's

©1981–2004, Deitel & Associates, Inc. All rights reserved.

sequence counter as shown in the following pseudocode. The value of the seqlock's sequence counter is initialized to zero.

```
Do
    Store value of seqlock's sequence counter in local variable seqTemp
    Execute instructions that read the value of the data protected by the seqlock
While seqTemp is odd or not equal to the value of the seqlock's sequence counter
```

After entering its critical section, the reader stores the value of the seqlock's sequence counter. Let us assume that this value is stored in variable `seqTemp`. The reader then accesses the data protected by the sequence counter. Consider what occurs if the system preempts the reader and a writer enters its critical section to modify the protected data. Before modifying the protected data, the writer must acquire the seqlock, which increments the value of the sequence counter. When the reader next executes, it compares the value stored in `seqTemp` to the current value of the sequence counter. If the two values are not equal, a writer must have entered its critical section. In this case, the value read by the reader may not be valid. Therefore, the loop continuation condition in the preceding pseudocode determines if the value of the sequence counter has changed since the reader accessed the protected data. If so, the reader continues the loop until it has read a valid copy of the protected data. Because the value of the seqlock's sequence counter is initialized to zero, a write is being performed when that value is odd. Thus, if `seqTemp` is odd, the reader will read the protected data while a writer is in the process of modifying it. In this case, the loop continuation condition ensures that the reader continues the loop to ensure that it reads valid data. When no writers have attempted to modify the data while the reader executes inside its critical section, the reader exits the `do...while` loop.

Because writers need not wait for readers to release a lock, seqlocks are appropriate for interrupt handling and other instances when writers must execute quickly to improve performance. In most cases, readers successfully read data on the first try. However, it is possible for readers to be indefinitely postponed while multiple writers modify shared data, so seqlocks should be used in situations where protected data is read more often than it is written.[191]

20.9.4 Kernel Semaphores

Spin locks and seqlocks perform well when the critical sections they protect contain few instructions. However, as the size of the critical section increases, the amount of time spent busy waiting increases, leading to significant overhead and performance degradation. Also, spin locks can lead to deadlock if a process sleeps while holding the lock. When a critical section must be protected for a long time, **kernel semaphores** are a better choice for implementing mutual exclusion. For example, only one process at a time should be able to read an image from a scanner. Because scanners may take several seconds to scan an image, a scanner device driver typically enforces mutually exclusive access to its scanner using semaphores.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Kernel semaphores are counting semaphores (see Section 5.6.3, Counting Semaphores) represented by a wait queue and a counter. The wait queue stores processes that are waiting on the kernel semaphore. The value of the counter determines how many processes may simultaneously access their critical sections. When a kernel semaphore is created, the counter is initialized to the number of processes allowed to access the semaphore concurrently. For example, if a semaphore protects access to three identical resources, then the initial counter value would be set to 3.[192]

When a process attempts to execute its critical section, it calls function `down` on the kernel semaphore. Function `down`, which corresponds to the *P* operation (see Section 5.6, Semaphores), checks the current value of the counter and responds according to the following rules:

- If the value of the counter is greater than 0, `down` decrements the counter and allows the process to execute.
- If the value of the counter is less than or equal to 0, `down` decrements the counter, and the process is added to the wait queue and enters the *sleeping* state. By putting a process to sleep, the kernel reduces the overhead due to busy waiting because sleeping processes are not dispatched to a processor.

When a process exits its critical section, it releases the kernel semaphore by calling function `up`. This function inspects the value of the counter and responds according to the following rules:

- If the value of the counter is greater than or equal to 0, `up` increments the counter.
- If the value of the counter is less than 0, `up` increments the counter, and a process from the wait queue is awakened so that it can execute its critical section.[193]

Kernel semaphores cause processes to sleep if placed in the semaphore's wait queue, so they cannot be used in interrupt handlers or when a spin lock is held. However, the kernel provides an alternative solution that allows interrupt handlers to access semaphores using function `down_trylock`. If the interrupt handler cannot enter the critical section protected by the kernel semaphore, function `down_trylock` will return to the caller instead of causing the interrupt handler to sleep.[194] Kernel semaphores should be used only for process that need to sleep while holding the semaphore; processes that sleep while holding a spin lock can lead to deadlock.

20.10 Interprocess Communication

Many of the interprocess communication (IPC) mechanisms available in Linux are derived from traditional UNIX IPC mechanisms, and they all have a common goal: to allow processes to exchange information. Although all IPC mechanisms accomplish this goal, some are better suited for particular applications, such as those that communicate over a network or exchange short messages with other local applications. In this section, we discuss how IPC mechanisms are implemented in the Linux kernel and how they are employed in Linux systems.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.10.1 Signals

Signals were one of the first interprocess communication mechanisms available in UNIX systems—the kernel uses them to notify processes when certain events occur. In contrast to the other Linux IPC mechanisms we discuss, signals do not allow processes to specify more than a word of data to exchange with other processes; signals are primarily intended to alert a process that an event has occurred.[195] The signals a Linux system supports depend on the processor architecture. Figure 20.24 lists the first 20 signals identified by the POSIX specification (all of today's architectures support these signals).[196]

Signals, which are created by the kernel in response to interrupts and exceptions, are sent to a process or thread either as a result of executing an instruction (such as a segmentation fault, `SIGSEGV`), from another process (such as when one process terminates another, `SIGKILL`) or from an asynchronous event (e.g., an I/O completion signal). The kernel delivers a signal to a process by pausing its execu-

Signal	Type	Default Action	Description
1	`SIGHUP`	Abort	Hang-up detected on terminal or death of controlling process
2	`SIGINT`	Abort	Interrupt from keyboard
3	`SIGQUIT`	Dump	Quit from keyboard
4	`SIGILL`	Dump	Illegal instruction
5	`SIGTRAP`	Dump	Trace/breakpoint trap
6	`SIGABRT`	Dump	Abort signal from `abort` function
7	`SIGBUS`	Dump	Bus error
8	`SIGFPE`	Dump	Floating point exception
9	`SIGKILL`	Abort	Kill signal
10	`SIGUSR1`	Abort	User-defined signal 1
11	`SIGSEGV`	Dump	Invalid memory reference
12	`SIGUSR2`	Abort	User-defined signal 2
13	`SIGPIPE`	Abort	Broken pipe: write to pipe with no readers
14	`SIGALRM`	Abort	Timer signal from `alarm` function
15	`SIGTERM`	Abort	Termination signal
16	`SIGSTKFLT`	Abort	Stack fault on coprocessor
17	`SIGCHLD`	Ignore	Child stopped or terminated
18	`SIGCONT`	Continue	Continue if stopped
19	`SIGSTOP`	Stop	Stop process
20	`SIGTSTP`	Stop	Stop typed at terminal device

Figure 20.24 | POSIX signals.[197]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

tion and invoking the process's corresponding signal handler. Once the signal handler completes execution, the process resumes execution.[198]

A process or thread can handle a signal in one of three ways. (1) Ignore the signal—processes can ignore all but the `SIGSTOP` and `SIGKILL` signals. (2) Catch the signal—when a process catches a signal, it invokes its signal handler to respond to the signal. (3) Execute the **default action** that the kernel defines for that signal—by default, the kernel defines one of five actions that is performed when a process receives a signal.[199]

The first default action is to abort, which causes the process to terminate immediately. The second is to perform a memory dump. A **memory dump** is similar to an abort; it causes a process to terminate, but before it does so, the process generates a **core file** that contains the process's execution context, which includes the process's stack, registers and other information useful for debugging. The third default action is simply to ignore the signal. The fourth is to stop (i.e., suspend) the process—often used to debug a process. The fifth is continue, which reverses the fourth by switching a process from the *suspended* state to the *ready* state.[200]

Processes can choose not to handle signals by **blocking** them. If a process blocks a specific signal type, the kernel does not deliver the signal until the process stops blocking it. Processes block a signal type by default while handling another signal of the same type. As a result, signal handlers need not be reentrant (unless the default behavior is not used), because multiple instances of the process's signal handler cannot be executed concurrently. It is, however, possible for a signal handler to interrupt a signal handler of a different type.[201]

Common signals, such as those shown in Fig. 20.24, are not queued by the kernel. If a signal is currently being handled by a process and a second signal of the same type is generated for that process, the kernel discards the latter signal. If two signals are generated simultaneously by an SMP system, the kernel simply drops one as a result of the race condition. In certain circumstances, dropped signals do not affect system behavior. For example, a single `SIGKILL` signal is sufficient for the system to terminate a process. In mission-critical systems, however, dropped signals could be disastrous. For example, a user-defined signal may be used to monitor safety systems that protect human life. If such signals were dropped, people's lives could be at risk. To ensure that such systems do not miss a signal, Linux supports **real-time signals**. These are queued by the kernel; therefore, multiple instances of the same signal can be generated simultaneously and not be discarded.[202] By default the kernel queues up to 1,024 real-time signals of the same type; any further signals are dropped.

20.10.2 Pipes

Pipes enable two processes to communicate using the producer/consumer model. The producer process writes data to the pipe, after which the consumer process reads data from the pipe in first-in-first-out order.

When a pipe is created, an inode is allocated and assigned to the pipe. Similar to procfs inodes (see Section 20.7.4, Proc File System), pipe inodes do not point to

©1981–2004, Deitel & Associates, Inc. All rights reserved.

disk blocks. Rather, they point to one page of data called a **pipe buffer** that the kernel uses as a circular buffer. Each pipe maintains a unique pipe buffer that stores data that is transferred between two processes.[204]

When pipes are created, the kernel allocates two file descriptors (see Section 20.7.1, Virtual File System) to allow access to the pipe: one for reading from the pipe and one for writing to the pipe. Pipes are represented by files and accessed via the virtual file system. To initiate communication using a pipe, one process must create the pipe, then fork a child process with which to communicate via the pipe. The `fork` system call enables pipe communication because it allows the child process to inherit the parent process's file descriptors. Alternatively, two processes can share a file descriptor using sockets, discuss in the section that follows. Although the kernel represents pipes as files, pipes cannot be accessed from the directory tree. This prevents a process from accessing a pipe unless it has obtained the pipe's file descriptor from the process that created it.[205]

One limitation of pipes is that they support communication only between processes that share file descriptors. Linux supports a variation of a pipe, called a **named pipe** or **FIFO**, that can be accessed via the directory tree. When a FIFO is created, its name is added to the directory tree. Processes can access the FIFO by pathname as they would any other file in the directory tree (the location and name of the file are typically known before the processes execute). Therefore, processes can communicate using a named pipe the same way they access data in a file system—by supplying the correct pathname and appropriate file permissions. However, unlike data files, FIFOs point to a buffer located in memory, not on disk. Therefore, FIFOs provide the simplicity of sharing data in files without the latency overhead created by disk access.[206] Another limitation of pipes that the fixed-size buffer can result in suboptimal performance if a producer and consumer work at different speeds, as discussed in Section 6.2.3, Monitor Example: Circular Buffer,

20.10.3 Sockets

The Linux **socket** IPC mechanism allows pairs of processes to exchange data by establishing direct bidirectional communication channels. Each process can use its socket to transfer data to and from another process. One limitation of pipes is that communication occurs in a single direction (from producer to consumer); however, many cooperating processes require bidirectional communication. In distributed systems, for example, processes may need to be able to send and receive remote procedure calls. In this case, pipes are insufficient, because they are limited to communication in one direction and are identified by file descriptors, which are not unique among multiple systems. To address this issue, sockets are designed allow communication between unrelated processes so that such processes can exchange information both locally and across a network. Because sockets allow such flexibility, they may perform worse than pipes in some situations. For example, if an application requires unidirectional communication between two processes in one system

©1981–2004, Deitel & Associates, Inc. All rights reserved.

(i.e., sending the output of a decompression program to a file on disk), pipes should be used instead of sockets.

There are two primary socket types that processes can employ. **Stream sockets** transfer information as streams of bytes. **Datagram sockets** transfer information in independent units called datagrams, discussed in Section 16.6.2, User Datagram Protocol (UDP).[207]

Stream Sockets

Processes that communicate via stream sockets follow the traditional client/server model. The server process creates a stream socket and listens for connection requests. A client process can then connect to the server process and begin exchanging information. Because data is transferred as a stream of bytes, processes communicating with stream sockets can read or write variable amounts of data. One useful property of stream sockets is that, unlike datagram sockets, they use TCP to communicate, which guarantees that all data transmitted will eventually arrive and will be delivered in the correct order. Because stream sockets inherently provide data integrity, they are typically the better choice when communication must be reliable.[208]

Datagram Sockets

Although stream sockets provide powerful IPC features, they are not always necessary or practical. Enabling reliable stream sockets creates more overhead than some applications can afford. Faster, but less reliable communication can be accomplished using datagram sockets. For example, in some distributed systems, a server with many clients periodically broadcasts status information to all of its clients. In this case, datagram sockets are preferable to stream sockets, because they require only a single message to be sent from the server socket and do not require any responses from clients. Datagrams may also be sent periodically to update client information, such as for clock synchronization purposes. In this case, each subsequent datagram is intended to replace the information contained in previous datagrams. Therefore, the clients can afford not to receive certain datagram packets, provided that future datagram packets arrive eventually. In such situations, where data loss is either unlikely or unimportant, applications can use datagram sockets in lieu of stream sockets to increase performance.

Socketpairs

Although sockets are most often used for Internet communication, Linux enables bidirectional communication between multiple processes on the same system using sockets. When a process creates a socket in the local system, it specifies a file name that is used as the socket's address. Other sockets on that system can use the file name to communicate with that socket by reading from, and writing to, a buffer. Like many data structures in the Linux kernel, sockets are stored internally as files, and therefore can be accessed via the virtual file system using the `read` and `write` system calls.[209]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Linux provides another IPC mechanism that is implemented using sockets, called a **socketpair**. A socketpair is a pair of connected, unnamed sockets. When a process creates a socketpair, the kernel creates two sockets, connects them, and returns a file descriptor for each socket.[210] Similar to pipes, unnamed sockets are limited to use by processes that share file descriptors. Socketpairs are traditionally employed when related processes require bidirectional communication.

20.10.4 Message Queues

Messages are an IPC mechanism to allow processes to transmit information that is composed of a message type and a variable-length data area. Message types are not defined by the kernel; when processes exchange information, they specify their own message types to distinguish between messages.

Messages are stored in **message queues**, where they remain until a process is ready to receive them. Message queues, unlike pipes, can be shared by processes that are not parent and child. When the kernel creates a message queue, it assigns it a unique identifier. Related processes can search for a message queue identifier in a global array of **message queue descriptors**. Each descriptor contains a queue of pending messages, a queue of processes waiting for messages (message receivers), a queue of processes waiting to send messages (message senders), and data describing the size and contents of the message queue.[211]

When a process adds a message to a message queue, the kernel checks the queue's list of receivers for a process waiting for messages of that type. If it finds any such processes, the kernel delivers the message each of them. If no receiver is waiting for a message of the specified type and enough space is available in the message queue, the kernel adds the message to a queue of pending messages of that type. If insufficient space is available, the message sender adds itself to the queue of message senders. Senders wait in this queue until space becomes available (i.e., when a message is removed from the queue of pending messages).[212]

When a process attempts to receive a message, the kernel searches for messages of a specified type in the appropriate message queue. If it finds such a message, the kernel removes the message from the queue and copies it to a buffer located in the address space of the process receiving the message. If no messages of the requested type are found, the process is added to the queue of message receivers where it waits until the requested type of message becomes available.[213]

20.10.5 Shared Memory

The primary advantage of shared memory over other forms of IPC is that, once a region of shared memory is established, access to memory is processed in user space and does not require the kernel to access the shared data. Thus, because processes do not invoke the kernel for each access to shared data, this type of IPC improves performance for processes that frequently access shared data. Another advantage of shared memory is that processes can share as much data as they can address, potentially eliminating the time spent waiting when a producer and consumer work

©1981–2004, Deitel & Associates, Inc. All rights reserved.

at different speeds using fixed-size buffers, as discussed in Section 6.2.3. Linux supports two standard interfaces to shared memory that are managed via tmpfs: System V and POSIX shared memory. [*Note*: Processes can also share memory using memory-mapped files.]

The Linux implementation of System V shared memory employs four standard system calls (Fig. 20.25). When a process has successfully allocated and attached a region of shared memory, it can reference data in that region as it would reference data using a pointer. The kernel maintains a unique identifier that describes the physical region of memory to which each shared memory segment belongs, deleting the shared memory region only when a process requests its deletion and when the number of processes to which it is attached is zero.[214]

POSIX shared memory requires the use of the system call `shm_open` to create a pointer to the region of shared memory and the system call `shm_unlink` to close the region. The shared memory region is stored as a file in the system's shared memory file system, which must be mounted at `/dev/shm`; in Linux, a tmpfs file system is typically mounted there (tmpfs is described in the next section). The `shm_open` call is analogous to opening a file, whereas the `shm_unlink` call is analogous to closing a link to a file. The file that represents the shared region is deleted when it is no longer attached to any processes.

Both System V and POSIX shared memory allow processes to share regions of memory and map that memory to each process's virtual address space. However, because POSIX shared memory does not allow processes to change privileges for shared segments, it is slightly less flexible than System V shared memory.[215] Neither POSIX nor System V shared memory provides any synchronization mechanisms to protect access to memory. If synchronization is required, processes typically employ semaphores.

Shared Memory Implementation

The goal of shared memory in an operating system is to provide access to shared data with low overhead while rigidly enforcing memory protection. Linux implements shared memory as a virtual memory area that is mapped to a region of physical memory. When a process attempts to access a shared memory region, the kernel

System V Shared Memory System Call	Purpose
`shmget`	Allocates a shared memory segment.
`shmat`	Attaches a shared memory segment to a process.
`shmctl`	Changes the shared memory segment's properties (e.g., permissions).
`shmdt`	Detaches (i.e., removes) a shared memory segment from a process.

Figure 20.25 | System V shared memory system calls.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

first determines if the process has permission to access it. If so, the kernel allocates a virtual memory area that is mapped to the region of shared memory, then attaches the virtual memory area to the process's virtual address space. The process may then access shared memory as it would any other memory in its virtual address space.

The kernel keeps track of shared memory usage by treating the region as a file in **tmpfs**, the **temporary file system**. Tmpfs has been designed to simplify shared memory management while maintaining good performance for the POSIX and System V shared memory specifications. As its name suggests, tmpfs is temporary, meaning that shared memory pages are not persistent. When a file in tmpfs is deleted, its page frames are freed. Tmpfs is also swappable; that is, data stored in the tmpfs can be swapped to the backing store when available memory becomes scarce. The page(s) containing the file can then be loaded from the backing store when referenced. This allows the system to fairly allocate page frames to all processes in the system. Tmpfs also reduces shared memory overhead because it does not require mounting or formatting for use. Finally, the kernel can set permissions of tmpfs files, which enables the kernel to implement the `shmctl` system call in System V shared memory.[216]

When the kernel is loaded, an instance of tmpfs is created. If the user wishes to mount a tmpfs file system to the local directory tree (which, as previously discussed, is required for POSIX shared memory), the user can mount a new instance of the file system and access it immediately. To further improve shared memory performance, tmpfs interfaces directly with the memory manager—it has minimal interaction with the virtual file system. Although tmpfs creates dentries, inodes and file structures that represent shared memory regions, generic VFS file operations are ignored in favor of tmpfs-specific routines that bypass the VFS layer. This relieves tmpfs of certain constraints typically imposed by the VFS (e.g., the VFS does not allow a file system to grow and shrink while it is mounted).[217]

20.10.6 System V Semaphores

Linux implements two types of semaphores: kernel semaphores (discussed in Section 20.9.4, Kernel Semaphores) and **System V semaphores**. Kernel semaphores are synchronization mechanisms employed throughout the kernel to protect critical sections. System V semaphores also protect critical sections and are implemented using similar mechanisms; however, they are designed for user processes to access via the system call interface. For the remainder of this discussion, we refer to System V semaphores simply as semaphores.

Because processes often need to protect a number of related resources, the kernel stores semaphores in **semaphore arrays**. Each semaphore in an array protects a particular resource.[218, 219]

Before a process can access resources protected by a semaphore array, the kernel requires that there be sufficient available resources to satisfy the process's request. Thus, semaphore arrays can be implemented as a deadlock prevention mechanism by denying Havender's "wait-for" condition (see Section 7.7.1, Denying

©1981–2004, Deitel & Associates, Inc. All rights reserved.

the "Wait-For" Condition). If a requested resource in semaphore array has been allocated to another process, the kernel blocks the requesting process and places its resource request in a queue of pending operations for that semaphore array. When a resource is returned to the semaphore array, the kernel examines the semaphore array's queue of pending operations for processes, and if a process can proceed, it is unblocked.[220]

To prevent deadlock from occurring if a process terminates prematurely while it holds resources controlled by a semaphore, the kernel tracks the operations each process performs on a semaphore. When a process exits, the kernel reverses all the semaphore operations it performed to allocate its resources. Finally, note that semaphore arrays offer no protection against indefinite postponement caused by poor programming in the user processes that access them.

20.11 Networking

The networking subsystem performs operations on network packets, each of which is stored in a contiguous physical memory area described by an `sk_buff` structure. As a packet traverses layers of the network subsystem, network protocols add and remove headers and trailers containing protocol-specific information (see Chapter 16, Introduction to Networking).[221]

20.11.1 Packet Processing

Figure 20.26 illustrates the path taken by network packets as they travel from a network interface card (NIC) through the kernel. When a NIC receives a packet, it issues an interrupt, which causes the NIC's interrupt handler to execute. The interrupt handler calls the network device's driver routine that allocates an `sk_buff` for the packet, then copies the packet from the network interface into the `sk_buff` and adds the packet to a queue of packets pending processing. A queue of pending packets is assigned to each processor; the interrupt handler assigns a packet to the queue belonging to the processor on which it executes.[222]

At this point, the packet resides in memory where it awaits further processing. Because interrupts are disabled while the top half of the interrupt handler executes, the kernel delays processing the packet. Instead, the interrupt handler raises a softirq to continue processing the packet. (Section 20.8.6, Interrupts, discussed softirqs.) After raising the softirq, the device driver routine returns and the interrupt handler exits.[223]

A single softirq processes all packets that the kernel receives on that processor. Because the kernel uses softirqs to process packets, network routines can execute concurrently on multiple processors in SMP systems, resulting in increased performance. When the scheduler dispatches the network softirq, the softirq processes packets in the processor's queue until either the queue is empty, a predefined maximum number of packets are processed or a time limit is reached. If one of the latter two conditions is met, the softirq is rescheduled and returns control of the processor to the scheduler.[224]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

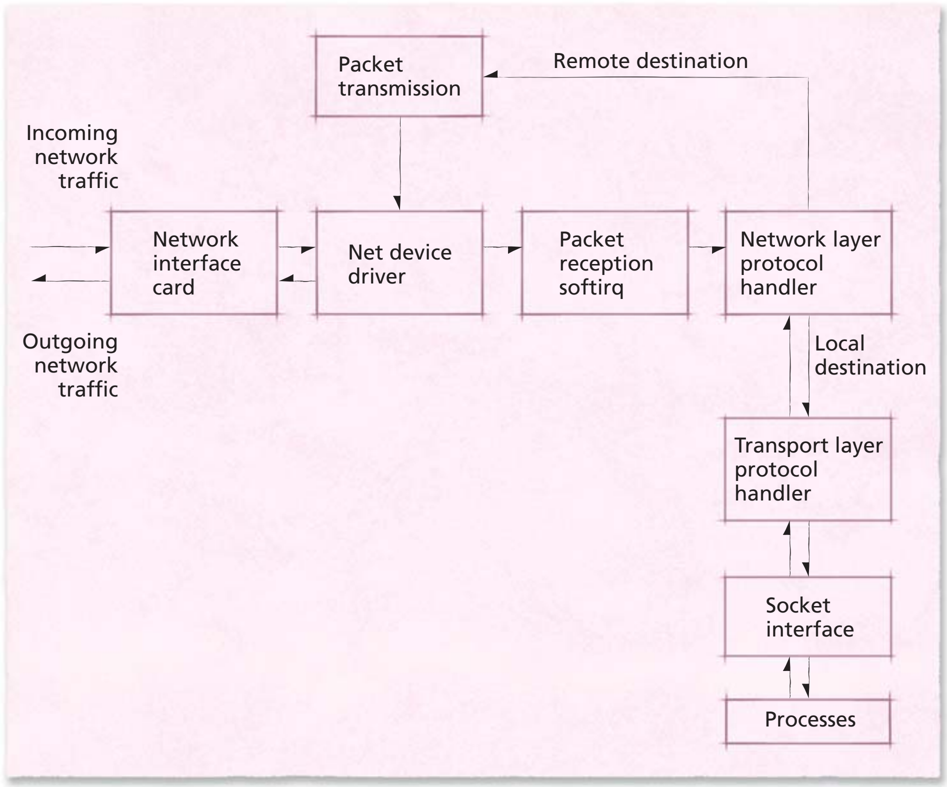


Figure 20.26 | *Path followed by network packets received by the networking subsystem.*

To process a packet, the network device softirq (called `NET_RX_SOFTIRQ`) removes the packet from the current processor's queue and passes it to the appropriate network layer protocol handler—typically the IP protocol handler. Although Linux supports other network layer protocols, they are rarely used. Therefore, we limit our discussion to the IP protocol handler.

When the IP protocol handler receives a packet, it first determines its destination. If the packet destination is another host, the handler forwards the packet to the appropriate host. If the packet is destined for the local machine, the IP protocol handler strips the IP protocol-specific header from the `sk_buff` and passes the packet to the appropriate transport layer packet handler. The transport layer packet handler supports the Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP).[225]

The transport layer packet handler determines the port specified by the TCP header and delivers the packet data to the socket that is bound to that port. The packet data is then transmitted to the process via the socket interface.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.11.2 Netfilter Framework and Hooks

As packets traverse the networking subsystem they encounter elements of the **netfilter** framework. Netfilter is a mechanism designed to allow kernel modules to directly inspect and modify packets. At various stages of the IP protocol handler, software constructs called **hooks** enable modules to register to examine, alter and/or discard packets. At each hook, modules can pass packets to user processes.[226]

Figure 20.27 lists the netfilter hooks and the packets that pass through each hook. The first hook packets encounter is `NF_IP_PRE_ROUTING`. All incoming packets pass it as they enter the IP protocol handler. One possible use for this hook is to enable a system that multiplexes network traffic (e.g., a load balancer). For example, a load balancer attempts to evenly distribute requests to a cluster of Web servers to improve average response times. Thus, the load balancer can register the `NF_IP_PRE_ROUTING` hook to intercept packets and reroute them according to the load on each Web server.

After a packet passes the `NF_IP_PRE_ROUTING` hook, the next hook it encounters depends on its destination. If the packet destination is the current network interface, it passes through the `NF_IP_LOCAL_IN` hook. Otherwise, if a packet needs to be passed to another network interface, it passes through the `NF_IP_FORWARD` hook. One possible use for these two hooks is to limit the amount of incoming traffic from a particular host by discarding packets once a certain threshold is reached.

All locally generated packets pass through the `NF_IP_LOCAL_OUT` hook, which, similar to the `NF_IP_LOCAL_IN` and `NF_IP_FORWARD` hooks, can be used to filter packets before they are sent across a network. Finally, immediately before leaving the system, all packets pass through the `NF_IP_POST_ROUTING` hook. A firewall can use this hook to modify the outgoing traffic to make it appear to have come from the firewall instead of from the original source.

20.12 Scalability

The early development of the Linux kernel focused on desktop systems and low-end servers. As additional features were implemented, Linux's popularity increased. This led to new interest in scaling Linux to larger systems (even mainframe computers) at large computer companies, such as IBM (`www.ibm.com`) and

Hook	Packets handled
`NF_IP_PRE_ROUTING`	All incoming packets.
`NF_IP_LOCAL_IN`	Packets sent to the local host.
`NF_IP_LOCAL_OUT`	Locally generated packets.
`NF_IP_FORWARD`	Packets forwarded to other hosts.
`NF_IP_POST_ROUTING`	All packets sent from the system.

Figure 20.27 | Netfilter hooks.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Hewlitt-Packard (www.hp.com), which cooperated with independent developers to scale Linux to be competitive in the high-end server market.

As Linux's scalability improves, designers must decide how far to enable the standard Linux kernel to scale. Increasing its scalability might negatively affect its performance on desktop systems and low-end servers. With this in mind, companies such as Red Hat (www.redhat.com), SuSE (www.suse.com) and Conectiva (www.conectiva.com) provide Linux distributions designed for high-performance servers and sell support services tailored to those distributions. By providing kernel modifications in a distribution, companies can tailor the Linux kernel to high-end servers without affecting users in other environments.

High-end servers are not the only reason to improve Linux's scalability—embedded-device manufacturers also use Linux to manage their systems. To satisfy the lesser needs of these limited-capability systems, software companies and independent developers create modified Linux kernels and applications designed for embedded devices. In addition to "home grown" embedded Linux kernels, companies such as Red Hat and projects such as uCLinux (www.uclinux.org) have developed Linux solutions for embedded systems. These modifications allow the kernel to execute in systems that do not support virtual memory, in addition to reducing resource consumption and enabling real-time execution. As developers continue to address these issues, Linux is becoming a viable choice for use in many embedded systems.

20.12.1 Symmetric Multiprocessing (SMP)

Much of the effort to increase Linux's performance on servers focuses on improved support for SMP systems. Version 2.0 was the first stable kernel release to support SMP systems.[227] Adding a global spin lock—called the **big kernel lock (BKL)**—was an early attempt at SMP support. When a process acquired the BKL in version 2.0, no process on any other processor could execute in kernel mode. Other processes were, however, free to execute in user mode.[228] The BKL enabled developers to serialize access to kernel data structures, which allowed the kernel to execute multiple processes concurrently in SMP systems. As one study showed, serializing access to the entire kernel meant that the system could not scale effectively to more than four processors per system.[229]

Locking the entire kernel is usually not required, because multiple processes can execute concurrently in kernel mode, provided they do not modify the same data structures. Linux kernel version 2.4 replaced most uses of the BKL with fine-grained locking mechanisms. This change allows SMP systems running Linux to scale effectively to 16 processors.[230]

Fine-grained locks, although a performance enhancement, tend to make developing and debugging the kernel more difficult. These locks force developers to carefully code the acquisition of the appropriate locks at the appropriate times to avoid causing deadlock (for example, by writing code that attempts to acquire the same spin lock twice). As a result, the use of fine-grained locks has slowed kernel development in many subsystems, as would be expected with the increased software complexity.[231]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Performance gains similar to those of large SMP systems have been achieved using alternative solutions, such as clustering (see Section 18.4, Clustering). Recall that a cluster of computer systems consists of several computers that cooperate to perform a common set of tasks. Such clusters of computers are typically connected by a dedicated, high-speed network. To perform work cooperatively using Linux systems, each system must run a modified kernel. Examples of features such a kernel might include are routines to balance workloads within the cluster and data structures to simplify remote interprocess communication, such as global process identifiers. If each machine in the Linux cluster contains a single processor, the complexity of fine-grained locks can be avoided, because only one kernel control path can execute at a time on each machine. Clusters can equal or outperform SMP systems, often at a lower cost of development and hardware.[232] For example, **Beowulf clusters**, a popular form of Linux clusters, have been used by NASA and the Department of Energy (DoE) to build high-performance systems at relatively low cost when compared to proprietary, multiprocessor supercomputer architectures.[233]

20.12.2 Nonuniform Memory Access (NUMA)

As the number of processors in high-end systems increases, buses that connect each processor to components such as memory become increasingly congested. Consequently, many system designers have implemented nonuniform memory access (NUMA) architectures to reduce the amount of bandwidth necessary to maintain high levels of performance in large multiprocessor systems. Recall from Section 15.4.2, Nonuniform Memory Access, that NUMA architectures divide a system into nodes, each of which provides high-performance interconnections between a set of processors, memory and/or I/O devices. The devices within a node are called local resources, while resources outside the node are called remote resources. Connections to remote resources are typically significantly slower than connections to local devices. To achieve high performance, the kernel must be aware of the layout of the NUMA system (i.e., the location and contents of each node) to reduce unnecessary internode access.

When the kernel detects a NUMA system, it must initially determine its layout (i.e., which devices correspond to which nodes) so that it can better allocate resources to processes. Most NUMA systems provide architecture-specific hardware that indicates the contents of each node. The kernel uses this information to assign each device an integer value indicating the node to which it belongs. Most NUMA systems partition a single physical memory address space into regions corresponding to each node. To support this feature, the kernel uses a data structure to associate a range of physical memory addresses with a particular node.[234]

To maximize performance, the kernel uses the layout of the NUMA system to allocate resources that are local to the node in which a process executes. For example, when a process is created on a particular processor, the kernel allocates the process local memory (i.e., memory that is assigned to the node containing the processor) using the data structure we mentioned. If a process were to subsequently

©1981–2004, Deitel & Associates, Inc. All rights reserved.

execute on a processor on a different node, its data would be stored in remote memory, leading to poor performance due to high memory access latency. Recall from Section 20.5.2, Process Scheduling, that the process scheduler will dispatch a process to the same processor (to improve cache performance) unless the number of processes running on each processor becomes unbalanced. To support NUMA systems, the load balancing routines attempt to migrate a process only to processors within the process's current node. In the case that a processor in another node is idle; however, the process scheduler will migrate a process to another node. Although this results in high memory access latency for the migrated process, overall system throughput increases due to increased resource utilization.[235]

Several other kernel components support the NUMA architecture. For example, when a process requests memory and available local memory is low, the kernel should swap out pages only in local memory. As a result, the kernel creates a *kswapd* thread for each node to perform page replacement.[236]

Despite the kernel's broad support for the NUMA architecture, there are limitations to the current implementation. For example, if a process is migrated to a processor on a remote node, the kernel provides no mechanisms to migrate the process's pages to local memory (i.e., memory in the process's current node). Only when pages are swapped from disk does the kernel move the process's pages to its processor's local memory. Several development projects exist to improve kernel support for NUMA systems, and extensions are expected to be released in future versions of the kernel.

20.12.3 Other Scalability Features

Developers have increased the size of several fields to accommodate the growing demands of computer systems. For example, the maximum number of users in a system increased from 16-bit field (65,536 users) to a 32-bit field (over four billion users). This is necessary for institutions such as large universities that have more than 100,000 users. Similarly, the number of tasks a system can execute increased from 32,000 to 4 million, which was necessary to support mainframe and other high-end systems, which often execute tens or hundreds of thousands of threads.[237] Also, the variable that stores time, `jiffies`, has been increased from a 32-bit to a 64-bit value. This means that the value, which is incremented at every timer interrupt, will not overflow at the current timer frequency (1,000Hz) for over 2 billion billion years. Using a 32-bit number, jiffies would overflow after approximately 50 days with a timer frequency of 1,000Hz.[238, 239, 240]

Fields related to storage also increased in size to accommodate large memories. For example, Linux can reference disk blocks using a 64-bit number, allowing the system to access 16 quintillion disk blocks—corresponding to exabytes (billions of gigabytes) of data. Linux also supports Intel's Physical Address Extension (PAE) technology, which allows systems to access up to 64GB of data (corresponding to a 36-bit address) using a 32-bit processor.[241, 242]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Prior to version 2.6, the kernel could not be preempted by a user process. However, the 2.6 kernel is preemptible, meaning that the kernel will be preempted if an event causes a high-priority task to be *ready*, which improves response times for real-time processes. To ensure mutual exclusion and atomic operations, the kernel disables preemption while executing a critical section. Finally, Linux includes support for several high-performance architectures, such as 64-bit processors (both the Intel Itanium processors, `www.intel.com/products/server/processors/server/itanium2/`, and AMD Opteron processors, `www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8825,00.html`) and Intel's HyperThreading technology (see `www.intel.com/info/hyperthreading/`).[243, 244]

20.12.4 Embedded Linux

Porting Linux to embedded devices introduces design challenges much different from those in SMP and NUMA systems. Embedded systems provide architectures with limited instruction sets, small memory and secondary storage sizes and devices that are not commonly found in desktops and workstations (e.g., touch-sensitive displays and device-specific input buttons). A variety of Linux distributions are tailored to meet the needs of embedded systems.

Often, providers of embedded Linux distributions must implement hard real-time process scheduling. Examples of systems requiring real-time embedded device management include cell phones, digital video recorders (e.g., TiVO; `www.tivo.com`) and network gateways.[245] To provide real-time execution in the Linux kernel, companies such as MontaVista Software (`www.mvista.com`) modify a few key components of the kernel. For example, developers must reduce scheduling overhead so that real-time process scheduling occurs quickly enough that the kernel meets real-time processes' timing constraints. The standard kernel's policy, although somewhat appropriate for real-time processes, is not sufficient for providing hard real-time guarantees (see Section 8.9, Real-Time Scheduling). This is because the Linux scheduler by default does not support deadlines. Embedded-device developers modify the scheduler to support additional priority levels, deadlines and lower scheduling latency.[246]

Other concerns specific to embedded systems also require modification to the Linux kernel. For example, some systems may include a relatively small amount of memory compared to desktop systems, so developers must reduce the size of the kernel footprint. Also, some embedded devices do not support virtual memory. As a result, the kernel must be modified to perform additional memory management operations (e.g., protection) in software.[247]

20.13 Security

The kernel provides a minimal set of security features, such as discretionary access control. Authentication is performed outside the kernel by user-level applications such as `login`. This simple security infrastructure has been designed to allow system administrators to redefine access control policies, customize the way Linux authenticates users and specify encryption algorithms that protect system resources.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.13.1 Authentication

By default, Linux authenticates its users by requiring them to provide a username and password via the `login` process. Each username is linked to an integer user ID. Passwords are hashed using the MD5 or DES algorithms, then stored in entries corresponding to user IDs in either the `/etc/passwd` or `/etc/shadow` file. [*Note*: Although DES was originally developed as an encryption algorithm, it can be used as a hash algorithm.] The choice of encryption algorithm and location of the password file can be modified by a system administrator. Unlike encryption algorithms, which can reverse the encryption operation using a decryption key, hash algorithms are not reversible. Consequently, Linux verifies a password entered at the login prompt by passing it through the hash algorithm and comparing it to the entry that corresponds to the user's ID number in the `/etc/passwd` or `/etc/shadow` file.[248, 249]

As discussed in Section 19.3.1, Basic Authentication, username and password authentication is susceptible to brute-force cracking, such as dictionary attacks. Linux addresses such problems by allowing system administrators to load **pluggable authentication modules (PAMs)**. These modules can reconfigure the system at run time to include enhanced authentication techniques. For example, the password system can be strengthened to disallow terms found in a dictionary and require users to choose new passwords regularly. PAM also supports smart card, Kerberos and voice authentication systems.[250] System administrators can use PAMs to select an authentication system that is most suitable for their environment without modifying the kernel or utility programs such as `login`.[251]

20.13.2 Access Control Methods

Linux secures system resources by controlling access to files. As described in Section 20.3, Linux Overview, the root user is given access to all resources in the system. To control how other users access resources, each file in the system is assigned **access control attributes** that specify file permissions and file attributes, as discussed in Section 20.7.3. In Linux, file permissions consist of a combination of read, write and/or execute permissions specified for three categories: *user*, *group* and *other*. The *user* file permissions are granted to the owner of the file. By default, a Linux file's owner is initially the user that created the file. *Group* permissions are applied if the user requesting the file is not the owner of the file, but is a member of *group*. Finally, *other* permissions are applied to users that are members of *user* neither nor *group*.[252]

File attributes are an additional security mechanism that is supported by some file systems (e.g., the ext2 file system). File attributes allow users to specify constraints on file access beyond read, write and execute. For example, the **append-only** file attribute specifies that any changes to the file must be appended to the end of the file. The **immutable** file attribute specifies that a file cannot be modified (including renaming and deletion) or linked (i.e., referenced using symbolic or hard links).[253, 254]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Linux Security Modules (LSM) Framework

In many environments, security provided by the default access control policy (i.e., discretionary access control) is insufficient. Thus, the kernel supports the **Linux security modules (LSM) framework** to allow a system administrator to customize the access control policy for a particular system using loadable kernel modules. To choose a different access control mechanism, system administrators need only install the kernel module that implements that mechanism. The kernel uses hooks inside the access control verification code to allow an LSM to enforce its access control policy. As a result, an LSM is invoked only if a process has been granted access to a resource via the default access control policy. If a process is denied access by the default access control policy, the registered LSM does not execute, reducing the overhead caused by an LSM.[255]

One popular LSM is `SELinux`, developed by the National Security Agency (NSA). `SELinux` replaces Linux's default discretionary access control policy with a **mandatory access control** (MAC) policy (see Section 19.4.2, Access Control Models and Policies). Such a policy allows the system administrator to set the security rules for all files; these rules cannot be overridden by malicious or inexperienced users. The disadvantages of MAC policies result from the need for a greater number of complex rules. More information about the LSM framework and modules such as `SELinux` can be found at the official LSM Web site (`lsm.immunix.org`).[256]

Privilege Inheritance and Capabilities

When a process is launched by a user, normally it executes with the same privileges as the user who launched it. It is sometimes necessary for users to execute applications with privileges other than those defined by their username and group. For example, many systems allow users to change their passwords using the `passwd` program. This program modifies the `/etc/passwd` or `/etc/shadow` file, which can be read by everyone, but written only with root privileges. To allow users to execute such programs, Linux provides the `setuid` and `setgid` permission bits. If the `setuid` permission bit for an executable file is set, the process that executes that file is assigned the same privileges as the owner of the file. Similarly, if the `setgid` permission bit for an executable file is set, the process that executes that file is assigned the same privileges as the group specified in the file attributes. Thus, users can modify the `/etc/password` file if the passwd program is owned by a user with root privileges and its `setuid` permission bit is set.[257]

Poorly written programs that modify the `setuid` or `setgid` bits can allow users access to sensitive data and system resources. To reduce the possibility of such a situation occurring, Linux provides the LSM Capabilities module to implement capabilities (see Section 19.4.3, Access Control Mechanisms). This allows Linux administrators greater flexibility in assigning access control privileges, such as the ability to assign privileges to applications rather than to users, which promotes fine-grained security.[258]

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.13.3 Cryptography

Although PAM and the LSM framework allow system administrators to create secure authentication systems and customized access control policies, they cannot protect data that is not controlled by the Linux kernel (e.g., data transmitted over a network or stored on disk). To enable users to access several forms of encryption to protect their data, Linux provides the **Cryptographic API**. Using this interface, processes can encrypt information using powerful algorithms such as DES, AES and MD5 (see Section 19.2, Cryptography). The kernel uses the Cryptographic API to implement secure network protocols such as IPSec (see Section 19.10, Secure Network Protocols).[259]

The Cryptographic API also allows users to create secure file systems without modifying the existing file system's code. To implement such a file system, encryption is implemented using a **loopback device** (Fig. 20.28), which is a layer between the virtual file system and the existing file system (e.g., ext2). When the virtual file system issues a read or write call, control passes to the loopback device. If the VFS issues a read call, the loopback device reads the requested (encrypted) data from the underlying file system. The loopback device then uses the Cryptographic API to decrypt the data and returns that data to the VFS. Similarly, the loopback device

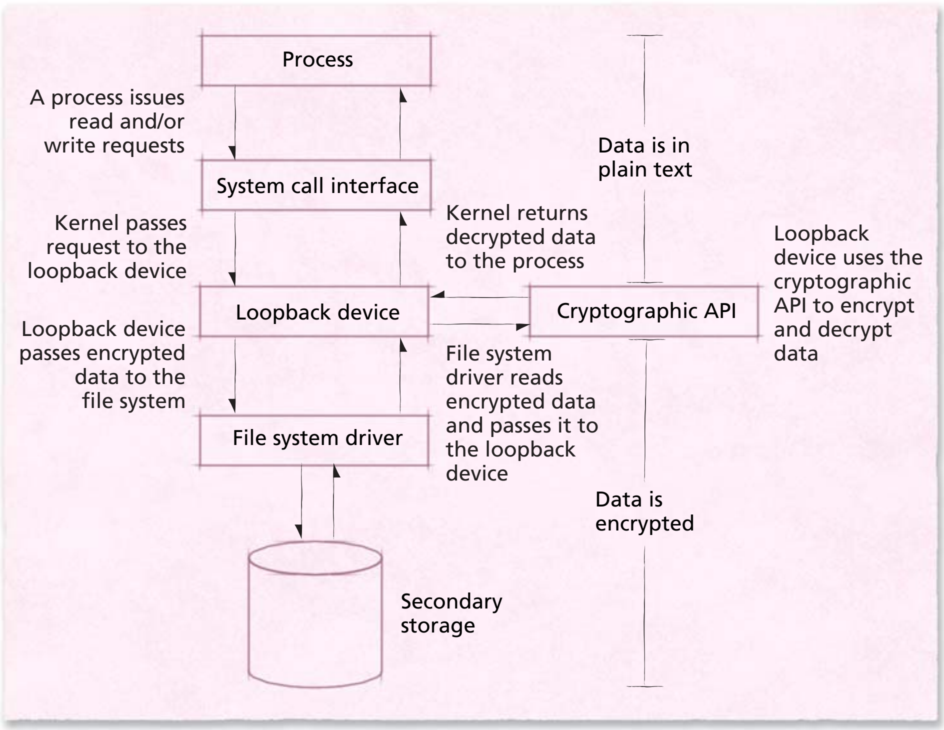

Process
A process issues read and/or write requests
System call interface
Kernel passes request to the loopback device
Loopback device
Loopback device passes encrypted data to the file system
File system driver
Secondary storage
Kernel returns decrypted data to the process
File system driver reads encrypted data and passes it to the loopback device
Cryptographic API
Loopback device uses the cryptographic API to encrypt and decrypt data
Data is in plain text
Data is encrypted

Figure 20.28 | Loopback device providing an encrypted file system using the Cryptographic API.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

uses the Cryptographic API to encrypt data before transferring it to the file system. This technique can be applied to individual directories or the entire file system, so that data is protected even if an unauthorized user accesses the hard disk using another operating system.[260]

Web Resources

`www.kernel.org/`
Official site for hosting the latest Linux kernel releases. It also includes a brief description of Linux and contains links to Linux resources.

`lxr.linux.no/`
Contains a cross reference to recent releases of the Linux source code. Users can navigate the source code specific to an architecture and version number, search the source for identifiers and other text, and compare two different versions of the source side by side. The `/Documentation` directory provides links to files referenced in the chapter.

`loll.sourceforge.net/linux/links/`
Provides an index of categorized links to a selection of online Linux resources. Categories include kernel, documentation, distributions, security and privacy, graphics and software applications.

`www.linux.org/`
Provides updated Linux information including news, development status and links to documentation, distributions, applications, source code and other resources.

`www.kernelnewbies.org/`
Provides information for people new to the Linux kernel. Features include a glossary, FAQ, articles, documentation, useful scripts and mailing lists.

`www.spinics.net/linux/`
Contains a large number of Linux resources, including links to information on kernels, security, embedded Linux and more.

`www.tldp.org/LDP/lki/index.html`
Part of the Linux Documentation Project hosted at `www.tldp.org` that discusses Linux kernel internals. The Linux Documentation Project provides many guides, HOWTOs and FAQs for those interested in the kernel, and for users and administrators.

`www.tldp.org/LDP/lkmpg/index.html`
Focuses on programming for the Linux environment, including details on programming kernel modules and adding entries to the proc file system.

`www.tldp.org/HOWTO/Module-HOWTO/`
Describes how to program Linux loadable kernel modules (LKMs) and discusses the use and behavior of LKMs.

`www.csn.ul.ie/~mel/projects/vm/`
Provides documentation of the Linux version 2.4 virtual memory system, which remains largely intact in version 2.6. The documents include a comprehensive discussion of Linux virtual memory and a companion code commentary.

`www.linuxjournal.com/search.php?query=&topic=4`
The *Linux Journal* Web site provides numerous articles on Linux. Be sure to search the “Linux Kernel” category.

`www.linuxsymposium.org/2003/`
Provides information about the Linux Symposia, annual gatherings of the top Linux kernel developers from over 30 countries.

`lse.sourceforge.net/`
Home to the Linux Scalability Effort, an organization dedicated to porting Linux to larger and more complex computer systems.

`www.linux-mag.com/`
Online version of *Linux Magazine*, which provides articles, guides and tutorials for Linux systems.

`www.linuxdevices.com/`
Provides articles and resources on Linux for embedded systems and discusses how Linux scales to devices such as cell phones and PDAs.

`www.linuxsecurity.com/`
Contains articles, documentation and other links to information regarding Linux security for both users and developers.

`kernelnewbies.org/documents/kdoc/kernel-locking/lklockingguide.html`
Contains information written by a kernel developer about kernel locks and semaphores.

`www.plig.org/xwinman/index.html`
Describes the window managers available for the X Window Manager as well as some of the desktop environments that run on X.

`user-mode-linux.sourceforge.net/`
Explains how to download, install and use User-Mode Linux (UML).

`lsm.immunix.org`
Provides links to documentation, downloads and mailing lists for the Linux Security Modules project.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

Key Terms

access control attribute (Linux)—Specifies the access rights for processes attempting to access a particular resource.

active list (Linux)—Scheduler structure that contains processes that will control the processor at least once during the current epoch.

active page (Linux)—Page of memory that will not be replaced the next time pages are selected for replacement.

***active* state** (Linux)—Task state describing tasks that can compete for execution on a processor during the current epoch.

append-only file attribute (Linux)—File attribute that limits users to appending data to existing file contents.

architecture-specific code—Code that specifies instructions unique to a particular architecture.

Beowulf cluster (Linux)—High-performance parallel-processing system consisting of a cluster of computers each running the Beowulf modification to the Linux kernel.

big kernel lock (BKL) (Linux)—Global spin lock that served as an early implementation of SMP support in the Linux kernel.

binary buddy algorithm (Linux)—Algorithm that Linux uses to allocate physical page frames. The algorithm maintains a list of groups of contiguous pages; the number in each group is a power of two. This facilitates memory allocation for processes and devices that require access to contiguous physical memory.

`bio` structure (Linux)—Structure that simplifies block I/O operations by mapping I/O requests to pages.

block allocation bitmap (Linux)—Bitmap that tracks the usage of blocks in each block group.

block group (Linux)—Collection of contiguous blocks managed by groupwide data structures so that related data blocks, inodes and other file system metadata are contiguous on disk.

bottom half of an interrupt handler (Linux)—Portion of interrupt-handling code that can be preempted.

bounce buffer (Linux)—Region of memory that allows the kernel to map data from the high memory zone into memory that it can directly reference. This is necessary when the system's physical address space is larger than kernel's virtual address space.

cache-cold process—Process that contains little, if any, of its data or instructions in the cache of the processor to which it will be dispatched.

cache-hot process—Process that contains most, if not all, of its data and instructions in the cache of the processor to which it will be dispatched.

capability—Security mechanism that assigns access rights to a subject (e.g., a process) by granting it a token for an object (i.e., a resource). This enables administrators to specify and enforce fine-grained access control.

code freeze (Linux)—Point at which no new code should be added to the kernel unless the code fixes a known bug.

core file (Linux)—File that contains the execution state of a process, typically used for debugging purposes after a process encounters a fatal exception.

Cryptographic API (Linux)—Kernel interface through which applications and services (e.g., file systems) can encrypt and decrypt data.

datagram socket—Socket that uses the UDP protocol to transmit data.

deadline scheduler (Linux)—Disk scheduling algorithm that eliminates indefinite postponement by assigning deadlines by which I/O requests are serviced.

daemon (Linux)—Process that runs periodically to perform system services.

deactivated process (Linux)—Process that has been removed from the run queues and can therefore no longer contend for processor time.

dcache (directory entry cache) (Linux)—Cache that stores directory entries (dentries), which enables the kernel to quickly map file descriptors to their corresponding inodes.

default action for a signal handler (Linux)—Predefined signal handler that is executed in response to a signal when a process does not specify a corresponding signal handler.

dentry (directory entry) (Linux)—Structure that maps a file to an inode.

desktop environment—GUI layer above a window manager that provides tools, applications and other software to improve system usability.

device class—Group of devices that perform similar functions.

device special file (Linux)—Entry in the `/dev` directory that provides access to a particular device.

direct I/O—Technique that performs I/O without using the kernel's buffer cache. This leads to more efficient memory utilization in database applications, which typically maintain their own buffer cache.

discretionary access control—Access control policy that specifies the owner of a file as the user that can assign access rights to that file.

distribution (Linux)—Software package containing the Linux kernel, user applications and/or tools that simplify the installation process.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

DMA memory (Linux)—Region of physical memory between zero and 16MB that is typically reserved for kernel bootstrapping code and legacy DMA devices.

doubly indirect pointer—Inode pointer that locates a block of (singly) indirect pointers.

effective priority (Linux)—Priority assigned to a process by adding its static priority to its priority boost or penalty.

epoch (Linux)—Time during which all processes move from the scheduler's active list to its expired list. This ensures that processes are not indefinitely postponed.

expired list (Linux)—Structure containing processes that cannot contend for the processor until the next epoch. Processes are placed in this list to prevent others from being indefinitely postponed. To quickly begin a new epoch, this list becomes the active list.

***expired* state** (Linux)—Task state that prevents a task from being dispatched until the next epoch.

ext2 inode (Linux)—Structure that stores information such as file size, the location of a file's data blocks and permissions for a single file or directory in an ext2 file system.

ext2fs (Linux)—Popular inode-based Linux file system that enables fast access to small files and supports large file sizes.

feature freeze (Linux)—State of kernel development during which no new features should be added to the kernel, in preparation for a new kernel release.

FIFO (Linux)—Named pipe that enables two unrelated processes to communicate via the producer/consumer relationship using a page-size buffer.

file attribute—File metadata that implements access control information, such as whether a file is append-only or immutable, that cannot be specified using standard Linux file permissions.

file permission—Structure that determines whether a user may read, write and or execute a file.

group descriptor (Linux)—Structure that records information regarding a block group, such as the locations of the inode allocation bitmap, block allocation bitmap and inode table.

high memory (Linux)—Region of physical memory (which begins at 896MB on the IA-32 architecture) beginning at the largest physical address that is permanently mapped to the kernel's virtual address space and extending to the limit of physical memory (64GB on Intel Pentium 4 processors). Because the kernel must perform expensive operations to map pages in its virtual address space to page frames in high memory, most kernel data structures are not stored in high memory.

hot swappable device—Device that can be added to, or removed from, a computer while it is running.

hook—Software feature that enables developers to add features to an existing application without modifying its source file. An application uses a hook to call a procedure that can be defined by another application.

immutable attribute (Linux)—Attribute specifying that a file can be read and executed, but cannot be copied, modified or deleted.

inactive list (Linux)—See expired list.

inactive page (Linux)—Page in main memory that can be replaced by an incoming page.

indirect pointer—Inode pointer that points to a block of inode pointers.

inode (Linux)—Structure that describes the location of data blocks corresponding to a file, directory or link in a file system. In the VFS, this structure represents any file in the system. An ext2 inode represents a file in the ext2 file system.

inode allocation bitmap (Linux)—Bitmap that records a block group's inode usage.

inode cache (Linux)—Cache that improves inode lookup performance.

inode table (Linux)—Structure that contains an entry for each allocated inode in a block group.

kernel control path (Linux)—A kernel execution context that may perform operations requiring mutual exclusive access to kernel data structures.

kernel semaphore (Linux)—Semaphore implemented by the kernel to provide mutual exclusion.

kernel thread (Linux)—Thread that executes kernel code.

ksoftirqd (Linux)—Daemon that schedules software interrupt handlers when softirq load is high.

kswapd (Linux)—Daemon that swaps pages to disk.

Linux security modules (LSM) framework (Linux)—Framework that allows system administrators to specify the access control mechanism employed by the system.

Linux Standard Base (Linux)—Project that aims to specify a standard Linux interface to improve application portability between kernel versions (and distributions).

loadable kernel module (Linux)—Software that can be integrated into the kernel at runtime.

load balancing—Operation that attempts to evenly distribute system load between processors in the system.

loopback device (Linux)—Virtual device that enables operations to be performed on data between layers of a system service (e.g., the file system).

©1981–2004, Deitel & Associates, Inc. All rights reserved.

mandatory access control—Access control policy that relegates assignment of access rights to the system administrator.

major device identification number (Linux)—Value that uniquely identifies a device in a particular device class. The kernel uses this value to determine a device's driver.

major version number (Linux)—Value that uniquely identifies a significant Linux release.

memory dump (Linux)—Action that generates a core file before terminating a process.

memory footprint (Linux)—Size of unswappable memory consumed by the kernel.

memory pool (Linux)—Region of memory reserved by the kernel for a process to ensure that the process's future requests for memory are not denied.

merge I/O requests (Linux)—To combine two I/O requests to adjacent locations on disk into a single request.

message queue (Linux)—Structure that stores messages that have yet to be delivered to processes.

message queue descriptor (Linux)—Structure that stores data regarding a message queue.

message—IPC mechanism that allows data to be transmitted by specifying a message type and variable-length field of data.

minor device identification number (Linux)—Value that uniquely identifies devices that are assigned the same major number (e.g., a hard drive partition).

minor version number (Linux)—Value that identifies successive stable (even) and development (odd) versions of the Linux kernel.

mount—Insert a file system into a local directory structure.

named pipe (Linux)—Pipe that can be accessed via the directory tree, enabling processes that are not parent and child to communicate using pipes. See also pipe.

netfilter framework (Linux)—Mechanism that allows kernel modules to directly inspect and modify packets. This is useful for applications such as firewalls, which modify each packet's source address before the packet is transmitted.

nice value (Linux)—Measure of a process's scheduling priority. Processes with a low nice value receive a greater share of processor time than other processes in the system and are therefore "less nice" to other processes in the system.

normal memory (Linux)—Physical memory locations beyond 16MB that the kernel can directly map to its virtual address space. This region is used to store kernel data and user pages.

package (Linux)—Portion of a distribution containing an application or service. Users can customize their Linux systems by adding and removing packages.

page cache (Linux)—Cache storing pages of data from disk. When a process requests data from disk, the kernel first determines if it exists in the page cache, which can eliminate an expensive disk I/O operation.

page global directory (Linux)—Virtual memory structure that stores addresses of second-level page-mapping tables.

page middle directory (Linux)—Virtual memory structure that stores addresses of third-level page-mapping tables (also called page tables).

page table (Linux)—Virtual memory structure that contains direct mappings between virtual page numbers and physical page numbers.

PID (process identifier)—Integer that uniquely identifies a process.

pipe—Interprocess communication mechanism that uses a page of memory as a first-in-first-out buffer to transfer information between processes.

pipe buffer (Linux)—Page of data that is used to buffer data written to a pipe.

pluggable authentication module (PAM) (Linux)—Module that can be installed at runtime to incorporate enhanced authentication techniques in the Linux system.

port of Linux—Version of the Linux kernel that is modified to support execution in a different environment.

preemption lock counter (Linux)—Integer that is used to determine whether code executing in kernel mode may be preempted. The value of the counter is incremented each time a kernel control path enters a critical section during which it cannot be preempted.

priority array (Linux)—Structure within a run queue that stores processes of the same priority.

procfs (proc file system) (Linux)—File system built directly into the kernel that provides real-time information about the status of the kernel and processes, such as memory utilization and system execution time.

ramfs (Linux)—Region of main memory treated as a block device. The ramfs file system must be formatted before use.

reader/writer lock (Linux)—Lock that allows multiple threads to concurrently hold a lock when reading from a resource, but only one thread to hold a lock when writing to that resource.

real-time signal (Linux)—Signal implementation that helps to implement a real-time system by ensuring that no signals are dropped.

request list (Linux)—Structure that stores pending I/O requests. This list is sorted to improve throughput by reducing seek times.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

reverse mapping (Linux)—Linked list of page table entries that reference a page of memory. This facilitates updating all PTEs corresponding to a shared page that is about to be replaced.

root user (Linux)—See superuser.

run queue (Linux)—List of processes waiting to execute on a particular processor.

second extended file system (ext2fs) (Linux)—See ext2fs.

semaphore array (Linux)—Linked list of semaphores that protect access to related resources.

seqlock (Linux)—Mutual exclusion structure that combines a spin lock with a sequence counter. Seqlocks are used by interrupt handlers, which require immediate exclusive access to data.

Single UNIX Specification—Specification (created by The Open Group) to which an operating system must conform to earn the right to display the UNIX trademark (see `www.unix.org/version3/overview.html`).

slab (Linux)—Page of memory that reduces internal fragmentation due to small structures by storing multiple structures smaller than one page.

slab allocator (Linux)—Kernel entity that allocates memory for objects placed in the slab cache.

slab cache (Linux)—Cache that stores recently used slabs.

socket—Interprocess communication mechanism that allows processes to exchange data by establishing direct communication channels. Enables processes to communicate over a network using `read` and `write` calls.

socketpair (Linux)—Pair of connected, unnamed sockets that can be used for bidirectional communication between processes on a single system.

socket address—Unique identifier for a socket.

software interrupt handler (Linux)—Interrupt-handling code that can be performed without masking interrupts and can therefore be preempted.

softirq (Linux)—Software interrupt handler that is reentrant and not serialized, so it can be executed on multiple processors simultaneously.

source tree (Linux)—Structure that contains source code files and directories. Provides a logical organization to the monolithic Linux kernel.

spin lock—Lock that provides mutually exclusive access to critical sections. When a process holding the lock is executing inside its critical section, any process concurrently executing on a different processor that attempts to acquire the lock before entering its critical section is made to busy wait.

starvation limit (Linux)—Time at which high-priority processes are placed in the expired list to prevent low-priority processes from being indefinitely postponed.

static priority level (Linux)—Integer value assigned to a process when it is created that determines its scheduling priority.

stream socket—Socket that transfers data using the TCP protocol.

superblock (Linux)—Block containing information regarding a mounted file system, such as the root inode and other information that protects the file system's integrity.

superuser (root user) (Linux)—User that may perform restricted operations (i.e., those that may damage the kernel and/or the system).

swap cache (Linux)—Cache of page table entries that describes whether a particular page exists in the system swap file on secondary storage. If a page table entry is present in the swap cache, then its corresponding page exists in the swap file and does not need to be written to the swap file.

system file system (sysfs) (Linux)—File system that allows processes to access structures defined by the unified device model.

task (Linux)—User execution context (i.e., process or thread) in Linux.

tasklet (Linux)—Software interrupt handler that cannot be executed simultaneously on multiple processors. Tasklets are used to execute nonreentrant bottom halves of interrupt handlers.

time slice (Linux)—Another term for quantum.

tmpfs (temporary file system) (Linux)—Similar to ramfs, but does not require formatting before use, meaning that the system can store files in the tmpfs without the organizational overhead typical of most file systems.

top half of an interrupt handler (Linux)—Nonpreemptible portion of interrupt-handling code that performs the minimum work required to acknowledge an interrupt before transferring execution to the preemptible bottom-half handler.

triply indirect pointer—Pointer in an inode that locates a block of doubly indirect pointers.

unified device model (Linux)—Internal device representation that relates devices to device drivers, device classes and system buses. The unified device model simplifies power management and hot swappable device management.

unmap a page (Linux)—To update page table entries to indicate that the corresponding page is no longer resident.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

User-Mode Linux (UML) (Linux)—Linux kernel that executes as a user process within a host Linux system.

virtual file system (Linux)—Interface that provides users with a common view of files and directories stored across multiple heterogeneous file systems.

virtual memory area (Linux)—Structure that describes a contiguous region of a process's virtual address space so that the kernel can perform operations on this region as a unit.

window manager (Linux)—Application that controls the placement, appearance, size and other attributes of windows in a GUI.

zone (memory) (Linux)—Region of physical memory. Linux divides main memory in the low, normal and high zones to allocate memory according to the architectural limitations of a system.

zone allocator—Memory subsystem that allocates pages from the zone to which it is assigned.

Exercises

20.1 [*Section 20.4.1, Hardware Platforms*] Describe several applications of User-Mode Linux (UML).

20.2 [*Section 20.4.2, Loadable Kernel Modules*] Why is it generally unsafe to load a kernel module written for kernel versions other than the current one?

20.3 [*Section 20.5.1, Process and Thread Organization*] Which threading model (see Section 4.6, Threading Models) can threads created using the `clone` system call implement? How do Linux threads differ from traditional threads? Discuss the benefits and drawbacks of this implementation.

20.4 [*Section 20.5.2, Process Scheduling*] How are the Linux process scheduler's run queues similar to multilevel feedback queues (see Section 8.7.6, Multilevel Feedback Queues)? How are they different?

20.5 [*Section 20.5.2, Process Scheduling*] Why should the Linux process scheduler penalize processor-bound processes?

20.6 [*Section 20.5.2, Process Scheduling*] Why does Linux prevent users without root privileges from creating real-time processes?

20.7 [*Section 20.6.1, Memory Organization*] Why does the kernel allocate page frames to processes from normal and high memory before allocating pages from DMA memory?

20.8 [*Section 20.6.1, Memory Organization*] What are the benefits and drawbacks of embedding the kernel virtual address space in each process's virtual address space?

20.9 [*Section 20.6.1, Memory Organization*] The x86-64 architecture uses four levels of page tables; each level contains 512 entries (using 64-bit PTEs). However, the kernel provides only three levels of page tables. Assuming that each PTE points to a 4KB page, what is the largest address space the kernel can allocate to processes in this architecture?

20.10 [*Section 20.6.2, Physical Memory Allocation and Deallocation*] How does the kernel reduce the amount of internal fragmentation caused by allocating memory to structures that are much smaller than a page?

20.11 [*Section 20.6.3, Page Replacement*] When is the kernel unable to immediately free a page frame to make room for an incoming page?

20.12 [*Section 20.6.4, Swapping*] When nonresident pages are retrieved from the backing store in Linux, the memory manager retrieves not only the requested page but also up to eight pages contiguous to it in the *running* process's virtual address space. Identify the type of prepaging implemented and describe the benefits and disadvantages of such a policy.

20.13 [*Section 20.7.1, Virtual File System*] List at least four different objects a VFS file can represent and describe the usage of each object.

20.14 [*Section 20.7.1, Virtual File System*] What role does the dentry object serve in the Linux VFS?

20.15 [*Section 20.7.2, Virtual File System Caches*] Is it possible for a file's inode to exist in the inode cache if its dentry is not located in the dentry cache?

20.16 [*Section 20.7.3, Second Extended File System (ext2fs)*] Compare and contrast the VFS and ext2 representation of an inode.

20.17 [*Section 20.7.3, Second Extended File System (ext2fs)*] Why do most file systems maintain redundant copies of their superblock throughout the disk?

20.18 [*Section 20.7.3, Second Extended File System (ext2fs)*] What are the primary contents of a block group and what purpose do they serve?

20.19 [*Section 20.7.4, Proc File System*] In what ways does the proc file system differ from a file system such as ext2fs?

20.20 [*Section 20.8.1, Device Drivers*] Explain the concept and usage of device special files.

20.21 [*Section 20.8.3, Block Device I/O*] Identify two mechanisms employed by the Linux block I/O subsystem that improve performance. Discuss how they improve performance and how they may, if ever, degrade performance.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

20.22 [*Section 20.8.4, Network Device I/O*] Compare and contrast networking I/O operations and block/character I/O operations.

20.23 [*Section 20.8.5, Unified Device Model*] How does the unified device model facilitate kernel support for hot swappable devices?

20.24 [*Section 20.8.6, Interrupts*] Why does the networking subsystem employ softirqs to process packets?

20.25 [*Section 20.9.1, Spin Locks*] What occurs if multiple kernel control paths concurrently attempt to acquire the same spin lock?

20.26 [*Section 20.10.1, Signals*] What problems can result from dropping a signal while a process handles a signal of the same type?

20.27 [*Section 20.10.4, Message Queues*] What could happen if a message's size is greater than a message queue's buffer?

20.28 [*Section 20.10.6, System V Semaphores*] Name a potential problem that occurs when processes are made to wait on a semaphore array.

20.29 [*Section 20.11.2, Netfilter Framework and Hooks*] What purpose does the netfilter framework serve?

20.30 [*Section 20.12.2, Nonuniform Memory Access (NUMA)*] Which kernel subsystems were modified to support NUMA?

20.31 [*Section 20.13.1, Authentication*] How does Linux protect user passwords from attackers, even if the attacker acquires the password file? How can this be circumvented?

20.32 [*Section 20.13.2, Access Control Methods*] Why might it be dangerous to set the `setuid` or `setgid` bits for an executable file?

Recommended Reading

Linux development is an ongoing process that includes contributions from developers worldwide. Because Torvalds frequently releases new kernel versions, some documentation can be outdated as soon as it is published. Hence, the most current information is usually found on the Web.

Useful resources include the magazines *Linux Journal* and *Linux Magazine*. Monthly issues cover a variety of Linux topics such as desktop and server applications, programming and, of course, the kernel. Selected articles and information about subscribing can be found at `www.linuxjournal.com` and `www.linux-mag.com`, respectively.

Readers looking to dig deeper into the Linux kernel should consider the book *Understanding the Linux Kernel,* 2nd ed., by Bovet and Cesati.[261] This book includes in-depth discussions of nearly all kernel subsystems. It explains kernel version 2.4—and, while some material has changed in kernel 2.6, much of the book's content is still relevant.

Another notable book is *Linux Device Drivers,* 2nd ed., by Rubini and Corbet.[262] The title can be misleading, as the book also explains a number of kernel subsystems of concern to device driver developers. It provides an in-depth explanation of the I/O subsystem as well as information about synchronization, memory management and networking. Like most Linux-related literature, the book is somewhat outdated compared to the most recent release of the kernel. The second edition discusses kernel version 2.4.

Works Cited

1. Kuwabara, K., "Linux: A Bazaar at the Edge of Chaos," *First Monday,* Vol. 5, No. 3, March 2000.
2. "Linux History," viewed July 8, 2003, <`www.li.org/linuxhistory.php`>.
3. "Linux History," viewed July 8, 2003, <`www.li.org/linuxhistory.php`>.
4. Torvalds, L., <`www2.educ.umu.se/~bjorn/linux/misc/linux-history.html`>.
5. Quinlan, D., "The Past and Future of Linux Standards," *Linux Journal,* Issue 62, June 1999.
6. Linux README, <`lxr.linux.no/source/README? v=1.0.9`>.
7. Wilburn, G., "Which Linux OS Is Best for You," *Computing Canada,* August 1999, p. 26.
8. Wheeler, D., "More Than a Gigabuck: Estimating GNU/Linux's Size," June 30, 2001 (updated July 29, 2002), version 1.07, <`www.dwheeler.com/sloc/`>.
9. McHugh, J., "Linux: The Making of a Global Hack," *Forbes Magazine,* August 1998.
10. Pranevich, J., "The Wonderful World of Linux 2.2," January 26, 1999, <`linuxtoday.com/news_story.php3?ltsn=1999-01-26-015-05-NW-SM`>.
11. McCarty, B., and P. McCarty, "Linux 2.4," January 2001, <`www.linux-mag.com/2001-01/linux24_01.html`>.
12. McCarty, B., and P. McCarty, "Linux 2.4," January 2001, <`www.linux-mag.com/2001-01/linux24_01.html`>.
13. "Whatever Happened to the Feature Freeze?" December 18, 2002, <`lwn.net/Articles/18454/`>.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

14. Index, <www.gnu.org>.
15. "LWN Distributions List," updated May 2003, <old.lwn.net/Distributions>.
16. <www.linux-mandrake.com>.
17. <www.redhat.com>.
18. <www.suse.com>.
19. <www.debian.org>.
20. <www.slackware.org>.
21. <www.uclinux.org>.
22. <www.zauruszone.com/wiki/index.php?OpenZaurus.org>.
23. <www.tldp.org/LDP/lfs/LFS/>.
24. slashdot.org/askslashdot/99/03/07/1357235.shtml.
25. Casha, R., "The Linux Terminal—A Beginners' Bash," November 8, 2001, <linux.org.mt/article/terminal>.
26. "CG252-502 X Windows: History of X," modified June 19, 1996, <nestroy.wi-inf.uni-essen.de/Lv/gui/cg252/course/lect4c1.html>.
27. Manrique, D., "X Window System Architecture Overview HOWTO," 2001, <www.linux.org/docs/ldp/howto/XWindow-Overview-HOWTO/>.
28. "The Single UNIX Specification, Version 3—Overview," modified January 27, 2002, <www.unix.org/version3/overview.html>.
29. "The Unix System Specification," <unix.org/what_is_unix/single_unix_specification.html>.
30. Linux Standard Base Specification 1.3, (c) 2000-2002 Free Standards Group, October 27, 2002.
31. Liedtke, J., "Toward Real Microkernels," *Communications of the ACM*, Vol. 39, No. 9, September 1996, p. 75.
32. "The Linux Kernel Archives," <www.kernel.org>.
33. Linux kernel source code, version 2.5.56, <www.kernel.org>.
34. "LinuxHQ: Distribution Links," <www.linuxhq.com/dist.html>.
35. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.
36. Welsh, M., "Implementing Loadable Kernel Modules for Linux," *Dr. Dobb's Journal,* May 1995, <www.ddj.com/articles/1995/9505/>.
37. Henderson, B., "Linux Loadable Kernel Module HOWTO," May 2002, <www.tldp.org/HOWTO/Module-HOWTO/>.
38. Henderson, B., "Linux Loadable Kernel Module HOWTO," May 2002, <www.tldp.org/HOWTO/Module-HOWTO/>.
39. Petersen, K., "Kmod: The Kernel Module Loader," Linux kernel source file, Linux/Documentation/kmod.txt <www.kernel.org>.
40. "The User-Mode Linux Kernel Home Page," July 23, 2003, <user-mode-linux.sourceforge.net>.
41. Dike, J., "A User-Mode Port of the Linux Kernel," August 25, 2000, <user-mode-linux.sourceforge.net/als2000/index.html>.
42. Aivazian, T., "Linux Kernel 2.4 Internals," August 23, 2001, <www.tldp.org/LDP/lki/lki.html>.
43. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.
44. SchlapBach, A., "Linux Process Scheduling," May 2, 2000, <iamexwiwww.unibe.ch/studenten/schlpbch/linuxScheduling/LinuxScheduling.htm>.
45. Aivazian, T., "Linux Kernel 2.4 Internals," August 23, 2001, <www.tldp.org/LDP/lki/lki.html>.
46. Walton, S., "Linux Threads Frequently Asked Questions," January 21, 1997, <www.tldp.org/FAQ/Threads-FAQ/>.
47. McCracken, D., "POSIX threads and the Linux Kernel," *Proceedings of the Ottawa Linux Symposium,* 2002, p. 332.
48. Arcomano, R., "KernelAnalysis-HOWTO," June 2, 2002, <www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html>.
49. Walton, S., "Linux Threads Frequently Asked Questions" <linas.org/linux/threads-faq.html>.
50. Drepper, U., and I. Molnar, "The Native POSIX Thread Library for Linux," January 30, 2003, <people.redhat.com/drepper/nptl-design.pdf>.
51. Molnar, I., Announcement to Linux mailing list, <lwn.net/2002/0110/a/scheduler.php3>.
52. Cross-Referencing Linux, <lxr.linux.no/source/include/asm-i386/param.h?v=2.6.0-test7#L5> and <lxr.linux.no/source/kernel/sched.c?v=2.6.0-test7#L1336>.
53. Cross-Referencing Linux, <lxr.linux.no/source/kernel/sched.c?v=2.5.56>.
54. Cross-Referencing Linux, <lxr.linux.no/source/kernel/sched.c?v=2.5.56>.
55. Linux kernel source code, version 2.6.0-test2, /kernel/sched.c, lines 80–106, <lxr.linux.no/source/kernel/sched.c?v=2.6.0-test2>.
56. Linux kernel source code, version 2.5.75, <lxr.linux.no/source/kernel/sched.c?v=2.5.75>.
57. Linux kernel source code, version 2.5.75, <lxr.linux.no/source/kernel/sched.c?v=2.5.75>.
58. Linux kernel source code, version 2.5.75, <lxr.linux.no/source/kernel/sched.c?v=2.5.75>.
59. Linux kernel source code, version 2.6.0-test2, <lxr.linux.no/source/arch/i386/mm/pageattr.c?v=2.6.0-test2>.
60. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/mm.h?v= 2.5.75>.
61. Eranian, S., and David Mosberger, "Virtual Memory in the IA-64 Linux Kernel," *informIT.com*, November 8, 2002, <www.informit.com/isapi/product_id~%7B79EC75E3-7AE9-4596-AF39-283490FAFCBD%7D/element_id~%7BCE3A6550-B6B6-44BA-B496-673E8337B5F4%7D/st~%7BBAC7BB78-22CD-4E1E-9387-19EEB5B71759%7D/session_id~%7B9E8FCA0D-31BA-42DD-AEBB-EC1617DE0EC7%7D/content/articlex.asp>.
62. Linux source code, <lxr.linux.no/source/include/asm-i386/page.h?v=2.5.56>.
63. Van Riel, R., "Page Replacement in Linux 2.4 Memory Management" <www.surriel.com/lectures/linux24-vm.html>.
64. Gorman, M., "Understanding the Linux Virtual Memory Manager," <www.csn.ul.ie/~mel/projects/vm/guide/html/understand/>.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

65. Van Riel, R., "Page Replacement in Linux 2.4 Memory Management" <www.surriel.com/lectures/linux24-vm.html>.

66. Linux kernel source code, version 2.5.56, <lxr.linux.no/source/include/linux/mmzone.h?v=2.5.56>.

67. Linux kernel source code, version 2.5.75, /Documentation/block/biodoc.txt.

68. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, 2001.

69. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

70. Gorman, M., "Understanding the Linux Virtual Memory Manager," <www.csn.ul.ie/~mel/projects/vm/guide/html/understand/>.

71. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/mm/page_alloc.c?v=2.5.75>.

72. Linux kernel source code, version 2.6.0-test2, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/mm.h?v=2.6.0-test2>.

73. Gorman, M., "Understanding the Linux Virtual Memory Manager," <www.csn.ul.ie/~mel/projects/vm/guide/html/understand/>.

74. Rusling, D., "The Linux Kernel," 1999 <www.tldp.org/LDP/tlk/tlk.html>.

75. Knowlton, K. C., "A Fast Storage Allocator," *Communications of the ACM,* Vol. 8, No. 10, October 1965, pp. 623–625.

76. Knuth, D. E., *The Art of Computer Programming,* Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968, pp. 435-455.

77. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

78. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

79. Nayani, A.; M. Gorman; and R. S. de Castro, "Memory Management in Linux: Desktop Companion to the Linux Source Code," May 25, 2002, <www.symonds.net/~abhi/files/mm/index.html>.

80. Gorman, M., "Slab Allocator," <www.csn.ul.ie/~mel/projects/vm/docs/slab.html>.

81. "Driver Porting: Low-Level Memory Allocation," LWN.net, February 2003, <lwn.net/Articles/22909/>.

82. Knapka, J., "Outline of the Linux Memory Management System," <home.earthlink.net/~jknapka/linux-mm/vmoutline.html>.

83. Knapka, J., "Outline of the Linux Memory Management System," <home.earthlink.net/~jknapka/linux-mm/vmoutline.html>.

84. Arcangeli, A., "Le novita' nel Kernel Linux," December 7, 2001, <old.lwn.net/2001/1213/aa-vm-talk/mgp00001.html>.

85. Arcangeli, A., "Le novita' nel Kernel Linux," December 7, 2001, <old.lwn.net/2001/1213/aa-vm-talk/mgp00001.html>.

86. Linux kernel source code, version 2.5.75, <www.kernel.org>.

87. Arcangeli, A., "Le novita' nel Kernel Linux," December 7, 2001, <old.lwn.net/2001/1213/aa-vm-talk/mgp00001.html>.

88. Linux kernel source code, version 2.5.75, <www.kernel.org>.

89. Arcangeli, A., "Le novita' nel Kernel Linux," December 7, 2001, <old.lwn.net/2001/1213/aa-vm-talk/mgp00001.html>.

90. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

91. Arcangeli, A., "Le novita' nel Kernel Linux," December 7, 2001, <old.lwn.net/2001/1213/aa-vm-talk/mgp00001.html>.

92. Corbet, J., "What Rik van Riel Is Up To," *Linux Weekly News,* January 24, 2002, <http//php.lwn.net/2002/0124/kernel.php3>.

93. Arcangeli, A., "Le novita' nel Kernel Linux," December 7, 2001, <old.lwn.net/2001/1213/aa-vm-talk/mgp00001.html>.

94. Linux source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/mm/page-writeback.c?v=2.5.75>.

95. Arcangeli, A., "Le novita' nel Kernel Linux," December 7, 2001, <old.lwn.net/2001/1213/aa-vm-talk/mgp00001.html>.

96. Linux kernel source code, version 2.5.56, <www.kernel.org>.

97. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

98. Brown, N., "The Linux Virtual File-System Layer," December 29, 1999, <www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html>.

99. Rubini, A., "The Virtual File System in Linux," *Linux Journal,* May 1997, <www.linuxjournal.com/print.php?side=2108>.

100. Brown, N., "The Linux Virtual File-System Layer," December 29, 1999, <www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html>.

101. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

102. Linux source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/fs.h?v=2.5.75>.

103. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

104. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/fs/dcache.c?v=2.5.75>.

105. Gooch, R., "Overview of the Virtual File System," July 1999, <www.atnf.csiro.au/people/rgooch/linux/vfs.txt>.

106. Linux kernel source code, version 2.5.56, <www.kernel.org>.

107. Brown, N., "The Linux Virtual File-System Layer," December 29, 1999, <www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html>.

108. Linux kernel source code, version 2.5.75, <http://miller.cs.wm.edu/lxr3.linux/http/source/fs/dcache.c?v=2.5.75>.

109. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

110. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/fs/namei.c?v=2.5.75>.

111. Brown, N., "The Linux Virtual File-System Layer," December 29, 1999, <www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html>.

112. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/fs/namei.c?v=2.5.75>.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

113. Card, R.; T. Ts'O; and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," <e2fsprogs.sourceforge.net/ext2intro.html>.

114. /Linux/Documentation/filesystems/ext2.txt, Linux kernel source, version 2.4.18, <www.kernel.org>.

115. Card, R.; T. Ts'O; and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," <e2fsprogs.sourceforge.net/ext2intro.html>.

116. /Linux/Documentation/filesystems/ext2.txt, Linux kernel source, version 2.5.75, <www.kernel.org>.

117. Card, R.; T. Ts'O; and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," <e2fsprogs.sourceforge.net/ext2intro.html>.

118. Appleton, R., "A Non-Technical Look Inside the Ext2 File System," *Linux Journal,* August 1997, <www.linuxjournal.com/print.php?sid=2151>.

119. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

120. Linux kernel source code, version 2.5.75, <www.kernel.org>.

121. Card, R.; T. Ts'O; and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," <e2fsprogs.sourceforge.net/ext2intro.html>.

122. Pranevich, J., "The Wonderful World of Linux 2.6," July 13, 2003, <www.kniggit.net/wwol26.html>.

123. Linux kernel source code, version 2.6.0-test2, /include/linux/ext2_fs.h, line 60, <lxr.linux.no/source/include/linux/ext2_fs.h?v=2.6.0-test2>.

124. Appleton, R., "A Non-Technical Look Inside the Ext2 File System," *Linux Journal,* August 1997, <www.linuxjournal.com/print.php?sid=2151>.

125. Bowden, T.; B. Bauer; and J. Nerin, "The /proc Filesystem," Linux kernel source file, Linux/Documentation/filesystems/proc.txt <www.kernel.org>.

126. Rubini, A., "The Virtual File System in Linux," *Linux Journal,* May 1997, <www.linuxjournal.com/print.php?side=2108>.

127. Linux kernel source code, version 2.5.75, <www.kernel.org>.

128. Mouw, E., "Linux Kernel Procfs Guide," June 2001, <www.kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfs-guide.html>.

129. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

130. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

131. <www.lanana.org/docs/device-list/devices.txt>.

132. "Linux Allocated Devices," <www.lanana.org/docs/device-list/devices.txt>.

133. Rubini, A., and J. Corbet, *Linux Device Drivers,* O'Reilly, 2001, pp. 55–57.

134. Matia, F., "Kernel Korner: Writing a Linux Driver," *Linux Journal,* April 1998, <www.linuxjournal.com/print.php?sid=2476>.

135. "The HyperNews Linux KHG Discussion Pages, Device Driver Basics," December 30, 1997, <users.evtek.fi/~tk/rt_html/ASICS.HTM>.

136. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

137. Zezschwitz, G., and A. Rubini, "Kernel Korner: The Devil's in the Details," *Linux Journal,* May 1996, <www.linuxjournal.com/article.php?sid=1221>.

138. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

139. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/fs/char_dev.c?v=2.5.7>.

140. Rusling, D., "The Linux Kernel," 1999, <www.tldp.org/LDP/tlk/tlk.html>.

141. Linux kernel source code, version 2.5.75, <www.kernel.org>.

142. Linux kernel source code, version 2.5.75, <www.kernel.org>.

143. Rubini, A., and J. Corbet, *Linux Device Drivers,* O'Reilly, 2001, 323–328, 334–338.

144. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/bio.h?v=2.5.75>.

145. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/bio.h?v=2.5.75>.

146. Gopinath, K.; N. Muppalaneni; N. Suresh Kumar; and P. Risbood, "A 3-Tier RAID Storage System with RAID1, RAID5 and Compressed RAID5 for Linux," *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference*, June 2000, pp. 18–23.

147. Love, R., "Interactive Kernel Performance," *Proceedings of the Linux Symposium*, 2003, pp. 306–307.

148. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/drivers/block/deadline-iosched.c?v=2.5.75>.

149. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/drivers/block/deadline-iosched.c?v=2.5.75>.

150. Axboe, J., "[PATCH] block/elevator updates + deadline i/o scheduler," Linux kernel mailing list, July 26, 2002.

151. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/drivers/block/deadline-iosched.c?v=2.5.75>.

152. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/drivers/block/deadline-iosched.c?v=2.5.75>.

153. Love, R., "Interactive Kernel Performance," *Proceedings of the Linux Symposium*, 2003, p. 308.

154. Iyer, S., and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *ACM SIGOPS Operating Systems Review, Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, Vol. 35, No. 5, October 2001.

155. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/drivers/block/as-iosched.c>.

156. Morton, A., "IO Scheduler Benchmarking," Linux kernel mailing list, February 20, 2003.

157. Linux kernel source code, version 2.5.75, <www.kernel.org>.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

158. Kalev, D., "Raw Disk I/O," October 2001, <www.itworld.com/nl/lnx_tip/10122001/pf_index.html>.

159. Rubini, A., and J. Corbet, *Linux Device Drivers,* O'Reilly, 2001, pp. 430–433.

160. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/netdevice.h?v=2.5.75>.

161. Adel, A., "Differentiated Services on Linux," <user.cs.tu-berlin.de/~adelhazm/study/diffserv.pdf>

162. Rubini, A., and J. Corbet, *Linux Device Drivers,* O'Reilly, 2001, pp. 445–448.

163. Corbet, J., "Porting Drivers to the 2.5 Kernel," *Proceedings of the Linux Symposium*, 2003, p. 149.

164. "Universal Serial Bus Specification," Compaq, Hewlitt-Packard, Intel, Lucent, Microsoft, NEC, Phillips, rev. 2.0, April 27, 2002, p. 18.

165. Corbet, J., "Driver Porting: Device Model Overview," <lwn.net/Articles/31185/>, May 2003.

166. Corbet, J., "Driver Porting: Device Model Overview," <lwn.net/Articles/31185/>, May 2003.

167. Mochel, P., "Sysfs—The Filesystem for Exporting Kernel Objects," Linux kernel source code, version 2.5.75, Documentation/filesystems/sysfs.txt, January 10, 2003.

168. Linux kernel source code, version 2.5.75, Documentation/driver-model/overview.txt.

169. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/device.h>.

170. Linux kernel source code, version 2.5.75, Documentation/driver-model/bus.txt.

171. Linux kernel source code, version 2.5.75, Documentation/driver-model/class.txt.

172. Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation, "Advanced Configuration and Power Management," rev. 2.0b, October 11, 2002, <www.acpi.info/spec.htm> p. 26.

173. Mochel, P., "Linux Kernel Power Management," *Proceedings of the Linux Symposium*, 2003, <archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Mochel-OLS2003.pdf>, pp. 344, 347.

174. Russell, P., "Unreliable Guide to Hacking the Linux Kernel," 2000, <www.netfilter.org/unreliable-guides/kernel-hacking/lk-hacking-guide.html>.

175. Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual*, Vol. 3, *System Programmer's Guide*, 2002, pp. 5–32

176. Russell, P., "Unreliable Guide to Hacking the Linux Kernel," 2000, <www.netfilter.org/unreliable-guides/kernel-hacking/lk-hacking-guide.html>.

177. Gatliff, W., "The Linux Kernel's Interrupt Controller API," 2001, <billgatliff.com/articles/emb-linux/interrupts.pdf>.

178. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/interrupt.h?v=2.5.75>.

179. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/interrupt.h?v=2.5.75>.

180. Gatliff, W., "The Linux Kernel's Interrupt Controller API," 2001, <billgatliff.com/articles/emb-linux/interrupts.pdf>.

181. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/kernel/softirq.c?v=2.5.75>.

182. Rubini, A., and J. Corbet, *Linux Device Drivers,* O'Reilly, 2001, p. 19.

183. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, 2001, pp. 300–301, 305–306, 523–532, 545.

184. Love, R., "Kernel Korner: Kernel Locking Techniques," *Linux Journal,* August 2002, <www.linuxjournal.com/article.php?sid=5833>.

185. Love, R., "Kernel Korner: Kernel Locking Techniques," *Linux Journal,* August 2002, <www.linuxjournal.com/article.php?sid=5833>.

186. Torvalds, L., "/Documentation/spinlocks.txt," Linux kernel source code, version 2.5.75, <www.kernel.org>.

187. Russell, P., "Unreliable Guide to Locking," 2000, <www.kernel-newbies.org/documents/kdoc/kernel-locking/lklocking-guide.html>.

188. Russell, P., "Unreliable Guide to Locking," 2000, <www.kernel-newbies.org/documents/kdoc/kernel-locking/lklocking-guide.html>.

189. Torvalds, L., "/Documentation/spinlocks.txt," Linux kernel source code, version 2.5.75, <www.kernel.org>.

190. "Driver Porting: Mutual Exclusion with Seqlocks," <lwn.net/Articles/22818/>.

191. "Driver Porting: Mutual Exclusion with Seqlocks," <lwn.net/Articles/22818/>.

192. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, 2001, pp. 305–306.

193. Love, R., "Kernel Korner: Kernel Locking Techniques," *Linux Journal,* August 2002, <www.linuxjournal.com/article.php?sid=5833>.

194. Love, R., "Kernel Korner: Kernel Locking Techniques," *Linux Journal,* August 2002, <www.linuxjournal.com/article.php?sid=5833>.

195. Bar, M., "Kernel Korner: The Linux Signals Handling Model," *Linux Journal,* May 2000, <www.linuxjournal.com/article.php?sid=3985>.

196. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/kernel/signal.c?v=2.5.75>.

197. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/kernel/signal.c?v=2.5.75>

198. Troan, E., "A Look at the Signal API," *Linux Magazine,* January 2000, <www.linux-mag.com/2000-01/compile_01.html>.

199. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, 2001, p. 253.

200. Bar, M., "Kernel Korner: The Linux Signals Handling Model," *Linux Journal,* May 2000, <www.linuxjournal.com/article.php?sid=3985>.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

201. Rusling, D., “The Linux Kernel,” 1999, <www.tldp.org/LDP/tlk/tlk.html>.

202. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O’Reilly, 2001, p. 253.

203. Linux kernel source code, version 2.6.0-test2, signal.c, line 38 <lxr.linux.no/source/kernel/signal.c?v=2.6.0-test2>

204. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O’Reilly, 2001, pp. 524–532.

205. Chelf, B., “Pipes and FIFOs,” *Linux Magazine,* January 2001, <www.linux-mag.com/2001-01/compile_01.html>.

206. Chelf, B., “Pipes and FIFOs,” *Linux Magazine,* January 2001, <www.linux-mag.com/2001-01/compile_01.html>.

207. Free Software Foundation, “The GNU C Library,” 1998, <www.gnu.org/manual/glibc-2.2.5/index.html>.

208. Sechrest, S., “An Introductory 4.4BSD Interprocess Communication Tutorial,” <docs.freebsd.org/44doc/psd/20.ipctut/paper.html>.

209. Free Software Foundation, “The GNU C Library,” 1998, <www.gnu.org/manual/glibc-2.2.5/index.html>.

210. Sechrest, S., “An Introductory 4.4BSD Interprocess Communication Tutorial,” <docs.freebsd.org/44doc/psd/20.ipctut/paper.html>.

211. Aivazian, T., “Linux Kernel 2.4 Internals,” August 23, 2001, <www.tldp.org/LDP/lki/lki.html>.

212. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O’Reilly, 2001, p. 545.

213. Aivazian, T., “Linux Kernel 2.4 Internals,” August 23, 2001 <www.tldp.org/LDP/lki/lki.html>.

214. Goldt, S., et al., “Shared Memory,” *The Linux Programmer's Guide,* <en.tldp.org/LDP/lpg/node65.html>, version 0.4, March 1995, and Linux source 2.5.56.

215. Linux man page: shm_open, <www.cwi.nl/~aeb/linux/man2html/man3/shm_open.3.html>.

216. Linux source, <lxr.linux.no/source/Documentation/filesystems/tmpfs.txt?v=2.5.56> and <lxr.linux.no/source/mm/shmem.c?v=2.5.56>

217. Linux source, <lxr.linux.no/source/Documentation/filesystems/tmpfs.txt? v=2.5.56> and <lxr.linux.no/source/mm/shmem.c?v=2.5.56>.

218. Aivazian, T., “Linux Kernel 2.4 Internals,” August 23, 2001, <www.tldp.org/LDP/lki/lki.html>.

219. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/ipc/sem.c?v=2.5.75>.

220. Aivazian, T., “Linux Kernel 2.4 Internals,” August 23, 2001, <www.tldp.org/LDP/lki/lki.html>.

221. Cox, A., “Network Buffers,” <www.linux.org.uk/Documents/buffers.html>.

222. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/net/core/dev.c?v=2.5.75>.

223. Welte, H., “The Journey of a Packet Through the Linux 2.4 Network Stack,” October 14, 2000, <www.gnumonks.org/ftp/pub/doc/packet-journey-2.4.html>.

224. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/net/core/dev.c?v=2.5.75>.

225. Dobbelaere, J., “Linux Kernel Internals: IP Network Layer,” 2001, <www.cs.wm.edu/~jdobbela/papers/ip.pdf>.

226. Welte, H., “The Netfilter Framework in Linux 2.4,” September 24, 2000, <www.gnumonks.org/papers/netfilter-lk2000/presentation.html>.

227. Schmidt, J., “Symmetrical Multiprocessing with Linux,” 1999, <www.heise.de/ct/english/98/13/140/>.

228. Tumenbayer, E., et al., “Linux SMP HOWTO,” July 9, 2002, <www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/SMP-HOWTO.pdf>.

229. Intel Corporation, “Linux Scalability: The Enterprise Question,” 2000, <www.intel.com/internetservices/intelsolutionservices/downloads/linux_scalability.pdf>.

230. Bergmann, K., “Linux for z/Series Performance Overview,” April 3, 2002, <www.linuxvm.org/present/SHARE98/S2561kba.pdf>.

231. McVoy, L., “SMP Scaling Considered Harmful,” July 22, 1999, <www.bitmover.com/llnl/smp.pdf>.

232. McVoy, L., “SMP Scaling Considered Harmful,” July 22, 1999, <www.bitmover.com/llnl/smp.pdf>.

233. Merkey, P., “Beowulf History,” <www.beowulf.org/beowulf/history.html>, viewed July 21, 2003.

234. Linux kernel source code, version 2.5.75, <miller.cs.wm.edu/lxr3.linux/http/source/include/linux/mmzone.h>.

235. Dobson, M.; P. Gaughen; M. Hohnbaum; and E. Focht, “Linux Support for NUMA Hardware,” *Proceedings of the Linux Symposium*, 2003, pp. 181–195.

236. Dobson, M.; P. Gaughen; M. Hohnbaum; and E. Focht, “Linux Support for NUMA Hardware,” *Proceedings of the Linux Symposium*, 2003, pp. 181–195.

237. Linux kernel source code, version 2.6.0-test7, <lxr.linux.no/source/include/linux/threads.h?v=2.6.0-test7#L33>.

238. Pranevich, J., “The Wonderful World of Linux 2.6,” July 13, 2003, <www.kniggit.net/wwol26.html>.

239. Linux kernel source code, version 2.5.75, <www.kernel.org>.

240. Corbet, J., “Driver Porting: Timekeeping Changes,” February 2003, <lwn.net/Articles/22808/>.

241. Pranevich, J., “The Wonderful World of Linux 2.6,” July 13, 2003, <www.kniggit.net/wwol26.html>.

242. Linux source kernel version 2.5.75 <www.kernel.org>.

243. J. Pranevich, “The Wonderful World of Linux 2.6,” July 13, 2003, <www.kniggit.net/wwol26.html>.

244. Linux kernel source code, version 2.5.75, <www.kernel.org>.

245. “The Embedded Linux ‘Cool Devices’ Quick Reference Guide,” modified March 21, 2002, <http://www.linuxdevices.com/articles/AT4936596231.html>.

246. “MontaVista Linux—Real-time Performance,” May 2002, MontaVista Software, <http://www.mvista.com/dswp/realtime.pdf>.

247. Lehrbaum, R., “Using Linux in Embedded Systems and Smart Devices,” viewed July 21, 2003, <www.linuxdevices.com/articles/AT3155773172.html>.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

248. Hatch, B., and J. Lee, *Hacking Linux Exposed*, McGraw-Hill: Osborne, 2003, pp. 384–386.

249. Toxen, B., *Real World Linux Security*, 2nd ed., Prentice Hall PTR, 2002.

250. "Modules/Applications Available or in Progress," modified May 31, 2003, <www.kernel.org/pub/linux/libs/pam/modules.html>.

251. Morgan, A., "The Linux-PAM Module Writers' Guide," May 9, 2002, <www.kernel.org/pub/linux/libs/pam/Linux-PAM-html/pam_modules.html>.

252. Hatch, B., and J. Lee, *Hacking Linux Exposed*, McGraw-Hill: Osborne, 2003, pp. 15–19.

253. Linux man pages, "CHATTR(1), change file attributes on a Linux second extended file system," <nodevice.com/cgi-bin/searchman?topic=chattr>.

254. Hatch, B., and J. Lee, *Hacking Linux Exposed*, McGraw-Hill: Osborne, 2003, p. 24.

255. Smalley, S.; T. Fraser; and C. Vance, "Linux Security Modules: General Security Hooks for Linux," <lsm.immunix.org/docs/overview/linuxsecuritymodule.html>.

256. Jaeger, T.; D. Safford; and H. Franke, "Security Requirements for the Deployment of the Linux Kernel in Enterprise Systems," <oss.software.ibm.com/linux/papers/security/les_whitepaper.pdf>.

257. Wheeler, D., "Secure Programming for Linux HOWTO," February 9, 2000, <www.theorygroup.com/Theory/FAQ/Secure-Programs-HOWTO.html>.

258. Wright, C., et al., "Linux Security Modules: General Security Support for the Linux Kernel," 2002, <lsm.immunix.org/docs/lsm-usenix-2002/html/>.

259. Bryson, D., "The Linux CryptoAPI: A User's Perspective," May 31, 2002, <www.kerneli.org/howto/index.php>.

260. Bryson, D., "Using CryptoAPI," May 31, 2002, <www.kerneli.org/howto/node3.php>.

261. Bovet, D., and M. Cesati, *Understanding the Linux Kernel,* O'Reilly, 2001.

262. Rubini, A., and J. Corbet, *Linux Device Drivers,* O'Reilly, 2001.

©1981–2004, Deitel & Associates, Inc. All rights reserved.

©1981–2004, Deitel & Associates, Inc. All rights reserved.