
Contents

PART ONE ■ OVERVIEW

Chapter 1 Introduction

1.1 What Operating Systems Do	4	1.8 Distributed Systems	35
1.2 Computer-System Organization	7	1.9 Kernel Data Structures	36
1.3 Computer-System Architecture	15	1.10 Computing Environments	40
1.4 Operating-System Operations	21	1.11 Free and Open-Source Operating Systems	46
1.5 Resource Management	27	Practice Exercises	53
1.6 Security and Protection	33	Further Reading	54
1.7 Virtualization	34		

Chapter 2 Operating-System Structures

2.1 Operating-System Services	55	2.7 Operating-System Design and Implementation	79
2.2 User and Operating-System Interface	58	2.8 Operating-System Structure	81
2.3 System Calls	62	2.9 Building and Booting an Operating System	92
2.4 System Services	74	2.10 Operating-System Debugging	95
2.5 Linkers and Loaders	75	2.11 Summary	100
2.6 Why Applications Are Operating-System Specific	77	Practice Exercises	101
		Further Reading	101

PART TWO ■ PROCESS MANAGEMENT

Chapter 3 Processes

3.1 Process Concept	106	3.7 Examples of IPC Systems	132
3.2 Process Scheduling	110	3.8 Communication in Client– Server Systems	145
3.3 Operations on Processes	116	3.9 Summary	153
3.4 Interprocess Communication	123	Practice Exercises	154
3.5 IPC in Shared-Memory Systems	125	Further Reading	156
3.6 IPC in Message-Passing Systems	127		

Chapter 4 Threads & Concurrency

4.1 Overview	160	4.6 Threading Issues	188
4.2 Multicore Programming	162	4.7 Operating-System Examples	194
4.3 Multithreading Models	166	4.8 Summary	196
4.4 Thread Libraries	168	Practice Exercises	197
4.5 Implicit Threading	176	Further Reading	198

Chapter 5 CPU Scheduling

5.1 Basic Concepts	200	5.7 Operating-System Examples	234
5.2 Scheduling Criteria	204	5.8 Algorithm Evaluation	244
5.3 Scheduling Algorithms	205	5.9 Summary	250
5.4 Thread Scheduling	217	Practice Exercises	251
5.5 Multi-Processor Scheduling	220	Further Reading	254
5.6 Real-Time CPU Scheduling	227		

PART THREE ■ PROCESS SYNCHRONIZATION**Chapter 6 Synchronization Tools**

6.1 Background	257	6.7 Monitors	276
6.2 The Critical-Section Problem	260	6.8 Liveness	283
6.3 Peterson's Solution	262	6.9 Evaluation	284
6.4 Hardware Support for Synchronization	265	6.10 Summary	286
6.5 Mutex Locks	270	Practice Exercises	287
6.6 Semaphores	272	Further Reading	288

Chapter 7 Synchronization Examples

7.1 Classic Problems of Synchronization	289	7.5 Alternative Approaches	311
7.2 Synchronization within the Kernel	295	7.6 Summary	314
7.3 POSIX Synchronization	299	Practice Exercises	314
7.4 Synchronization in Java	303	Further Reading	315

Chapter 8 Deadlocks

8.1 System Model	318	8.6 Deadlock Avoidance	330
8.2 Deadlock in Multithreaded Applications	319	8.7 Deadlock Detection	337
8.3 Deadlock Characterization	321	8.8 Recovery from Deadlock	341
8.4 Methods for Handling Deadlocks	326	8.9 Summary	343
8.5 Deadlock Prevention	327	Practice Exercises	344
		Further Reading	346

PART FOUR ■ MEMORY MANAGEMENT

Chapter 9 Main Memory

9.1 Background	349	9.6 Example: Intel 32- and 64-bit Architectures	379
9.2 Contiguous Memory Allocation	356	9.7 Example: ARMv8 Architecture	383
9.3 Paging	360	9.8 Summary	384
9.4 Structure of the Page Table	371	Practice Exercises	385
9.5 Swapping	376	Further Reading	387

Chapter 10 Virtual Memory

10.1 Background	389	10.8 Allocating Kernel Memory	426
10.2 Demand Paging	392	10.9 Other Considerations	430
10.3 Copy-on-Write	399	10.10 Operating-System Examples	436
10.4 Page Replacement	401	10.11 Summary	440
10.5 Allocation of Frames	413	Practice Exercises	441
10.6 Thrashing	419	Further Reading	444
10.7 Memory Compression	425		

PART FIVE ■ STORAGE MANAGEMENT

Chapter 11 Mass-Storage Structure

11.1 Overview of Mass-Storage Structure	449	11.6 Swap-Space Management	467
11.2 HDD Scheduling	457	11.7 Storage Attachment	469
11.3 NVM Scheduling	461	11.8 RAID Structure	473
11.4 Error Detection and Correction	462	11.9 Summary	485
11.5 Storage Device Management	463	Practice Exercises	486
		Further Reading	487

Chapter 12 I/O Systems

12.1 Overview	489	12.6 STREAMS	519
12.2 I/O Hardware	490	12.7 Performance	521
12.3 Application I/O Interface	500	12.8 Summary	524
12.4 Kernel I/O Subsystem	508	Practice Exercises	525
12.5 Transforming I/O Requests to Hardware Operations	516	Further Reading	526

PART SIX ■ FILE SYSTEM**Chapter 13 File-System Interface**

13.1 File Concept	529	13.5 Memory-Mapped Files	555
13.2 Access Methods	539	13.6 Summary	560
13.3 Directory Structure	541	Practice Exercises	560
13.4 Protection	550	Further Reading	561

Chapter 14 File-System Implementation

14.1 File-System Structure	564	14.7 Recovery	586
14.2 File-System Operations	566	14.8 Example: The WAFL File System	589
14.3 Directory Implementation	568	14.9 Summary	593
14.4 Allocation Methods	570	Practice Exercises	594
14.5 Free-Space Management	578	Further Reading	594
14.6 Efficiency and Performance	582		

Chapter 15 File-System Internals

15.1 File Systems	597	15.7 Consistency Semantics	608
15.2 File-System Mounting	598	15.8 NFS	610
15.3 Partitions and Mounting	601	15.9 Summary	615
15.4 File Sharing	602	Practice Exercises	616
15.5 Virtual File Systems	603	Further Reading	617
15.6 Remote File Systems	605		

PART SEVEN ■ SECURITY AND PROTECTION**Chapter 16 Security**

16.1 The Security Problem	621	16.6 Implementing Security Defenses	653
16.2 Program Threats	625	16.7 An Example: Windows 10	662
16.3 System and Network Threats	634	16.8 Summary	664
16.4 Cryptography as a Security Tool	637	Further Reading	665
16.5 User Authentication	648		

Chapter 17 Protection

17.1 Goals of Protection	667	17.9 Mandatory Access Control (MAC)	684
17.2 Principles of Protection	668	17.10 Capability-Based Systems	685
17.3 Protection Rings	669	17.11 Other Protection Improvement Methods	687
17.4 Domain of Protection	671	17.12 Language-Based Protection	690
17.5 Access Matrix	675	17.13 Summary	696
17.6 Implementation of the Access Matrix	679	Further Reading	697
17.7 Revocation of Access Rights	682		
17.8 Role-Based Access Control	683		

Introduction



An **operating system** is software that manages a computer's hardware. It also provides a basis for application programs and acts as an **intermediary** between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks in a wide **variety of computing environments**. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing environments.

In order to explore the role of an operating system in a modern computing environment, it is important first to understand the **organization and architecture of computer hardware**. This includes the **CPU, memory, and I/O devices**, as well as storage. A fundamental responsibility of an operating system is to **allocate these resources to programs**.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter, we provide a general overview of the major components of a contemporary computer system as well as the functions provided by the operating system. Additionally, we cover several topics to help set the stage for the remainder of the text: data structures used in operating systems, computing environments, and open-source and free operating systems.

CHAPTER OBJECTIVES

- Describe the general organization of a computer system and the role of interrupts.
- Describe the components in a modern multiprocessor computer system.
- Illustrate the transition from user mode to kernel mode.
- Discuss how operating systems are used in various computing environments.
- Provide examples of free and open-source operating systems.

1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the **hardware**, the **operating system**, the **application programs**, and a **user** (Figure 1.1).

The **hardware**—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an **environment** within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

1.1.1 User View

The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and security and none paid to **resource utilization**—how various hardware and software resources are shared.

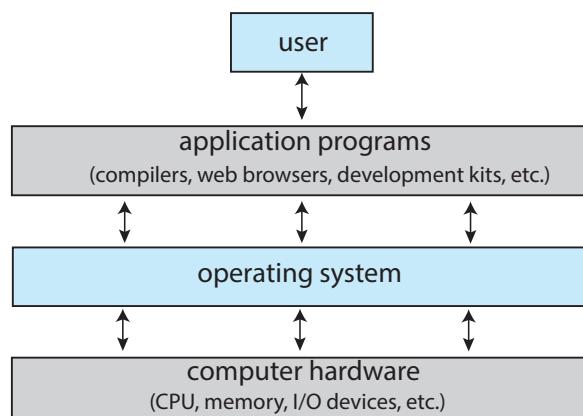


Figure 1.1 Abstract view of the components of a computer system.

Increasingly, many users interact with mobile devices such as smartphones and tablets—devices that are replacing desktop and laptop computer systems for some users. These devices are typically connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a **voice recognition** interface, such as Apple’s **Siri**.

Some computers have little or no user view. For example, **embedded computers** in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems and applications are designed primarily to run without user intervention.

1.1.2 System View

From the computer’s point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

1.1.3 Defining Operating Systems

By now, you can probably see that the term *operating system* covers many roles and functions. That is the case, at least in part, because of the myriad designs and uses of computers. Computers are present within toasters, cars, ships, spacecraft, homes, and businesses. They are the basis for game machines, cable TV tuners, and industrial control systems.

To explain this diversity, we can turn to the history of computers. Although computers have a relatively short history, they have evolved rapidly. Computing started as an experiment to determine what could be done and quickly moved to fixed-purpose systems for military uses, such as code breaking and trajectory plotting, and governmental uses, such as census calculation. Those early computers evolved into general-purpose, multifunction mainframes, and that’s when operating systems were born. In the 1960s, **Moore’s Law** predicted that the number of transistors on an integrated circuit would double every 18 months, and that prediction has held true. Computers gained in functionality and shrank in size, leading to a vast number of uses and a vast number and variety of operating systems. (See Appendix A for more details on the history of operating systems.)

How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems

exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order “the operating system.” The features included, however, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the **kernel**. Along with the kernel, there are two other types of programs: **system programs**, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

The matter of what constitutes an operating system became increasingly important as personal computers became more widespread and operating systems grew increasingly sophisticated. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. (For example, a web browser was an integral part of Microsoft’s operating systems.) As a result, Microsoft was found guilty of using its operating-system monopoly to limit competition.

Today, however, if we look at operating systems for mobile devices, we see that once again the number of features constituting the operating system is increasing. Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple’s iOS and Google’s Android—features

WHY STUDY OPERATING SYSTEMS?

Although there are many practitioners of computer science, only a small percentage of them will be involved in the creation or modification of an operating system. Why, then, study operating systems and how they work? Simply because, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming. Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

In summary, for our purposes, the operating system includes the always-running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while it is running. Most of this text is concerned with the kernel of general-purpose operating systems, but other components are discussed as needed to fully explain operating system design and operation.

1.2 Computer-System Organization

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common **bus** that provides access between components and shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display). Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

In the following subsections, we describe some basics of how such a system operates, focusing on three key aspects of the system. We start with interrupts, which alert the CPU to events that require attention. We then discuss storage structure and I/O structure.

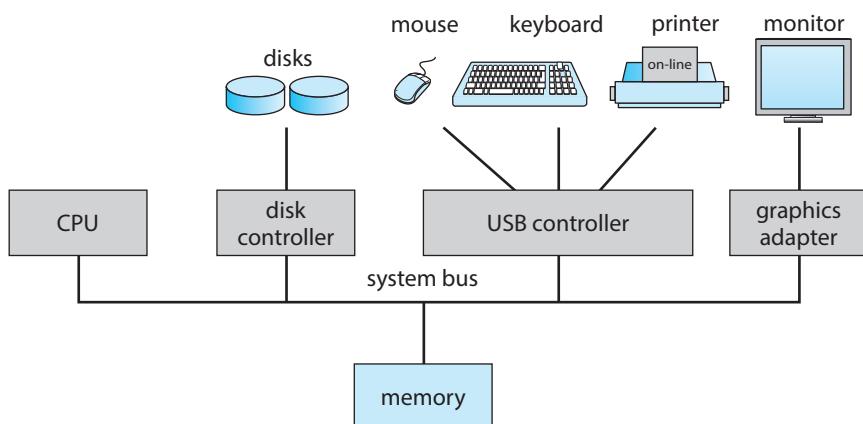


Figure 1.2 A typical PC computer system.

1.2.1 Interrupts

Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation. The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as “write completed successfully” or “device busy”. But how does the controller inform the device driver that it has finished its operation? This is accomplished via an **interrupt**.

1.2.1.1 Overview

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. (There may be many buses within a computer system, but the system bus is the main communications path between the major components.) Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of this operation is shown in Figure 1.3. To run the animation associated with this figure please click [here](#).

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for managing this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn,

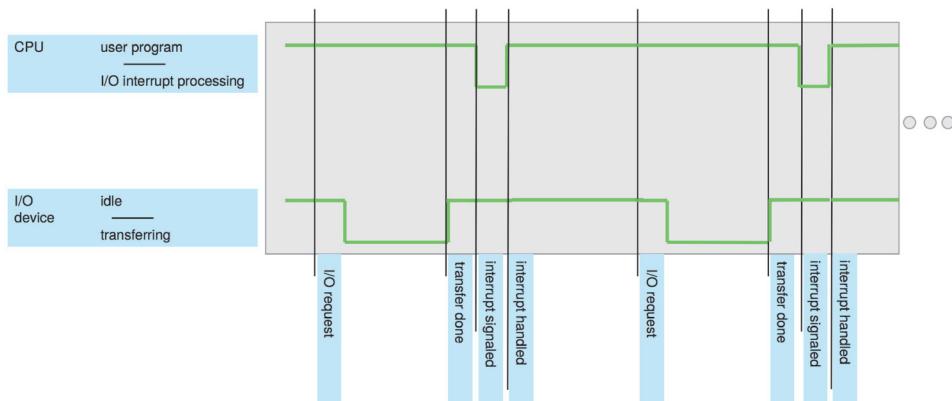


Figure 1.3 Interrupt timeline for a single program doing output.

would call the interrupt-specific handler. However, interrupts must be handled quickly, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

1.2.1.2 Implementation

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector. It then starts execution at the address associated with that index. The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a `return_from_interrupt` instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller *raises* an interrupt by asserting a signal on the interrupt request line, the CPU *catches* the interrupt and *dispatches* it to the interrupt handler, and the handler *clears* the interrupt by servicing the device. Figure 1.4 summarizes the interrupt-driven I/O cycle.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and the **interrupt-controller hardware**.

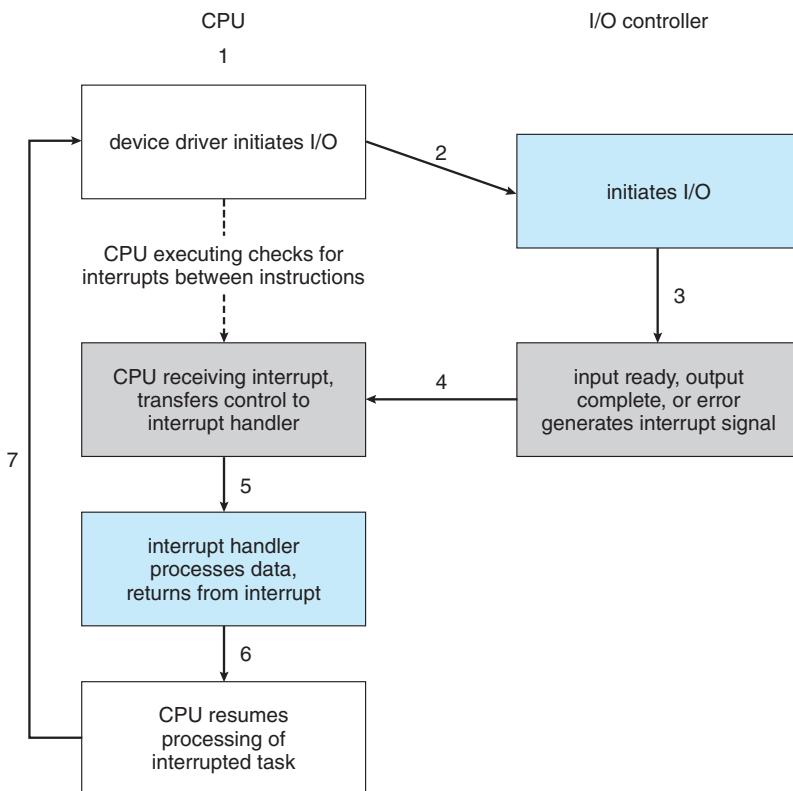


Figure 1.4 Interrupt-driven I/O cycle.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

Recall that the purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 1.5 illustrates the design of the interrupt vector for Intel processors. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority inter-

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 1.5 Intel processor event-vector table.

rupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events (and for other purposes we will discuss throughout the text). Device controllers and hardware faults raise interrupts. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

1.2.2 Storage Structure

The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**.

Computers use other forms of memory as well. For example, the first program to run on computer power-on is a **bootstrap program**, which then loads the operating system. Since RAM is **volatile**—loses its content when power is turned off or otherwise lost—we cannot trust it to hold the bootstrap program. Instead, for this and some other purposes, the computer uses electrically erasable programmable read-only memory (EEPROM) and other forms of **firmware**—storage that is infrequently written to and is nonvolatile. EEPROM

STORAGE DEFINITIONS AND NOTATION

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or **KB**, is 1,024 bytes; a **megabyte**, or **MB**, is $1,024^2$ bytes; a **gigabyte**, or **GB**, is $1,024^3$ bytes; a **terabyte**, or **TB**, is $1,024^4$ bytes; and a **petabyte**, or **PB**, is $1,024^5$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.

A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore **how** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible on most systems for two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory, as mentioned, is volatile—it loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage devices are **hard-disk drives (HDDs)** and **nonvolatile memory (NVM) devices**, which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing. Secondary storage is also much slower than main memory. Hence, the proper management of secondary storage is of central importance to a computer system, as we discuss in Chapter 11.

In a larger sense, however, the storage structure that we have described—consisting of registers, main memory, and secondary storage—is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes—to store backup copies of material stored on other devices, for example—are called **tertiary storage**. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.6) according to storage capacity and access time. As a general rule, there is a

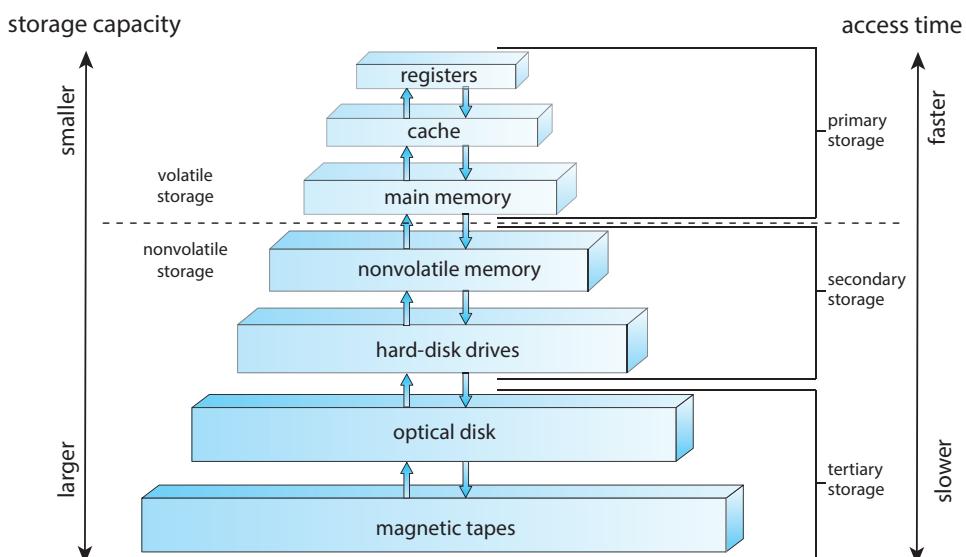


Figure 1.6 Storage-device hierarchy.

trade-off between size and speed, with smaller and faster memory closer to the CPU. As shown in the figure, in addition to differing in speed and capacity, the various storage systems are either volatile or nonvolatile. Volatile storage, as mentioned earlier, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

The top four levels of memory in the figure are constructed using **semiconductor memory**, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

Since storage plays an important role in operating-system structure, we will refer to it frequently in the text. In general, we will use the following terminology:

- Volatile storage will be referred to simply as **memory**. If we need to emphasize a particular type of storage device (for example, a register), we will do so explicitly.
- Nonvolatile storage retains its contents when power is lost. It will be referred to as **NVS**. The vast majority of the time we spend on NVS will be on secondary storage. This type of storage can be classified into two distinct types:
 - **Mechanical**. A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape. If we need to emphasize a particular type of mechanical storage device (for example, magnetic tape), we will do so explicitly.
 - **Electrical**. A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD. Electrical storage will be referred to as **NVM**. If we need to emphasize a particular type of electrical storage device (for example, SSD), we will do so explicitly.

Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

The design of a complete storage system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile storage as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

1.2.3 I/O Structure

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

Recall from the beginning of this section that a general-purpose computer system consists of multiple devices, all of which exchange data via a common

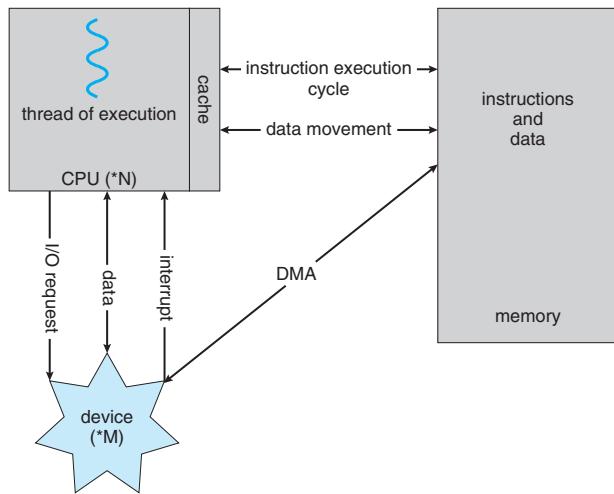


Figure 1.7 How a modern computer system works.

bus. The form of interrupt-driven I/O described in Section 1.2.1 is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as NVS I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.7 shows the interplay of all components of a computer system.

1.3 Computer-System Architecture

In Section 1.2, we introduced the general structure of a typical computer system. A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

1.3.1 Single-Processor Systems

Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The **core** is the component that executes instructions and registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. These systems have other special-purpose proces-

sors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers.

All of these special-purpose processors run a limited instruction set and do not run processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU core and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU with a single processing core, then the system is a single-processor system. According to this definition, however, very few contemporary computer systems are single-processor systems.

1.3.2 Multiprocessor Systems

On modern computers, from mobile devices to servers, **multiprocessor systems** now dominate the landscape of computing. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The primary advantage of multiprocessor systems is increased throughput. That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N , however; it is less than N . When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

The most common multiprocessor systems use **symmetric multiprocessing (SMP)**, in which each peer CPU processor performs all tasks, including operating-system functions and user processes. Figure 1.8 illustrates a typical SMP architecture with two processors, each with its own CPU. Notice that each CPU processor has its own set of registers, as well as a private—or local—cache. However, all processors share physical memory over the system bus.

The benefit of this model is that many processes can run simultaneously— N processes can run if there are N CPUs—without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory—to be shared dynamically among the various processors and can lower the workload variance among the processors. Such a system must be written carefully, as we shall see in Chapter 5 and Chapter 6.

The definition of *multiprocessor* has evolved over time and now includes **multicore** systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.

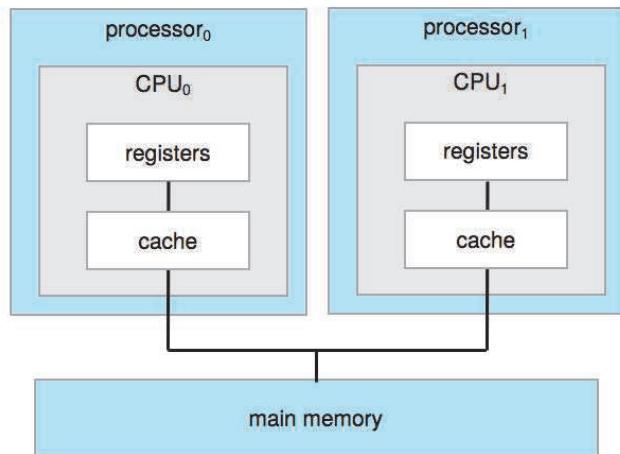


Figure 1.8 Symmetric multiprocessing architecture.

In addition, one chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for mobile devices as well as laptops.

In Figure 1.9, we show a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache. Notice, too, that a level 2 (L2) cache is local to the chip but is shared by the two processing cores. Most architectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared

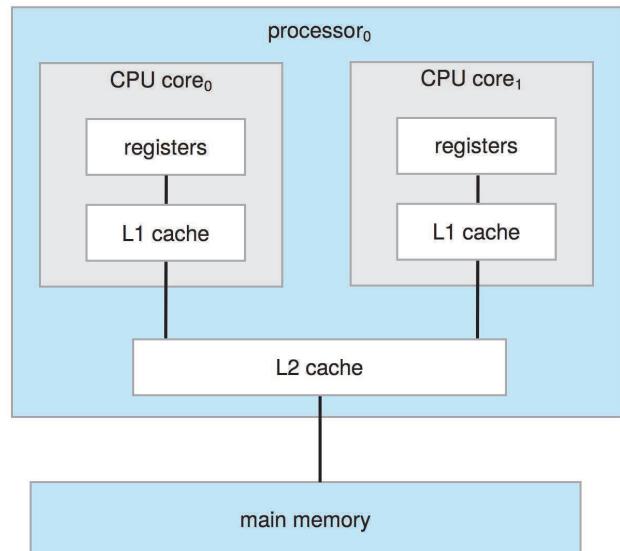


Figure 1.9 A dual-core design with two cores on the same chip.

DEFINITIONS OF COMPUTER SYSTEM COMPONENTS

- **CPU**—The hardware that executes instructions.
- **Processor**—A physical chip that contains one or more CPUs.
- **Core**—The basic computation unit of the CPU.
- **Multicore**—Including multiple computing cores on the same CPU.
- **Multiprocessor**—Including multiple processors.

Although virtually all systems are now multicore, we use the general term *CPU* when referring to a single computational unit of a computer system and *core* as well as *multicore* when specifically referring to one or more cores on a CPU.

caches. Aside from architectural considerations, such as cache, memory, and bus contention, a multicore processor with N cores appears to the operating system as N standard CPUs. This characteristic puts pressure on operating-system designers—and application programmers—to make efficient use of these processing cores, an issue we pursue in Chapter 4. Virtually all modern operating systems—including Windows, macOS, and Linux, as well as Android and iOS mobile systems—support multicore SMP systems.

Adding additional CPUs to a multiprocessor system will increase computing power; however, as suggested earlier, the concept does not scale very well, and once we add too many CPUs, contention for the system bus becomes a bottleneck and performance begins to degrade. An alternative approach is instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus. The CPUs are connected by a **shared system interconnect**, so that all CPUs share one physical address space. This approach—known as **non-uniform memory access**, or **NUMA**—is illustrated in Figure 1.10. The advantage is that, when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

A potential drawback with a NUMA system is increased latency when a CPU must access remote memory across the system interconnect, creating a possible performance penalty. In other words, for example, CPU_0 cannot access the local memory of CPU_3 as quickly as it can access its own local memory, slowing down performance. Operating systems can minimize this NUMA penalty through careful CPU scheduling and memory management, as discussed in Section 5.5.2 and Section 10.5.4. Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

Finally, **blade servers** are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between

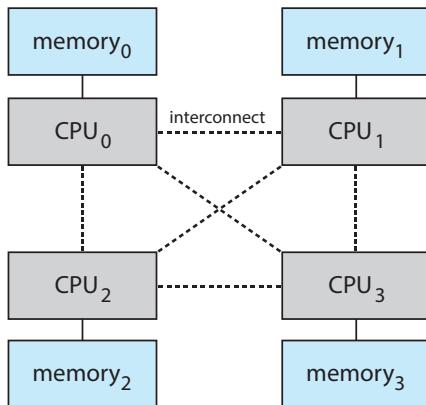


Figure 1.10 NUMA multiprocessing architecture.

types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

1.3.3 Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems described in Section 1.3.2 in that they are composed of two or more individual systems—or nodes—joined together; each node is typically a multicore system. Such systems are considered **loosely coupled**. We should note that the definition of *clustered* is not concrete; many commercial and open-source packages wrestle to define what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN (as described in Chapter 19) or a faster interconnect, such as InfiniBand.

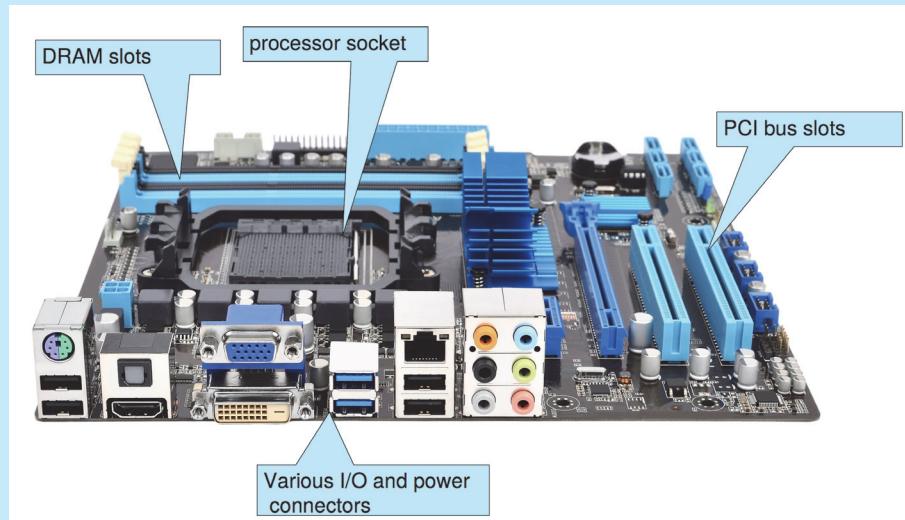
Clustering is usually used to provide **high-availability service**—that is, service that will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the network). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

High availability provides increased reliability, which is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active

PC MOTHERBOARD

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

server. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However, it does require that more than one application be available to run.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual cores in a computer or computers in a cluster. Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Chapter 19). Parallel clusters allow multiple hosts to access the same data on shared storage. Because most oper-

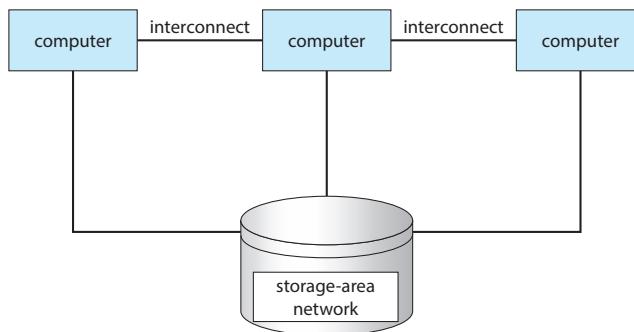


Figure 1.11 General structure of a clustered system.

ating systems lack support for simultaneous data access by multiple hosts, parallel clusters usually require the use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager (DLM)**, is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support thousands of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs)**, as described in Section 11.7.4, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.11 depicts the general structure of a clustered system.

1.4 Operating-System Operations

Now that we have discussed basic information about computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. As noted earlier, this initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to

HADOOP

Hadoop is an open-source software framework that is used for distributed processing of large data sets (known as **big data**) in a clustered system containing simple, low-cost hardware components. Hadoop is designed to scale from a single system to a cluster containing thousands of computing nodes. Tasks are assigned to a node in the cluster, and Hadoop arranges communication between nodes to manage parallel computations to process and coalesce results. Hadoop also detects and manages failures in nodes, providing an efficient and highly reliable distributed computing service.

Hadoop is organized around the following three components:

1. A distributed file system that manages data and files across distributed computing nodes.
2. The YARN (“Yet Another Resource Negotiator”) framework, which manages resources within the cluster as well as scheduling tasks on nodes in the cluster.
3. The **MapReduce** system, which allows parallel processing of data across nodes in the cluster.

Hadoop is designed to run on Linux systems, and Hadoop applications can be written using several programming languages, including scripting languages such as PHP, Perl, and Python. Java is a popular choice for developing Hadoop applications, as Hadoop has several Java libraries that support MapReduce. More information on MapReduce and Hadoop can be found at https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html and <https://hadoop.apache.org>

start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel by system programs that are loaded into memory at boot time to become **system daemons**, which run the entire time the kernel is running. On Linux, the first system program is “`systemd`,” and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt. In Section 1.2.1 we described hardware interrupts. Another form of interrupt is a **trap** (or an **exception**), which is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed by executing a special operation called a **system call**.

1.4.1 Multiprogramming and Multitasking

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Furthermore, users typically *want* to run more than one program at a time as well. **Multiprogramming** increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a **process**.

The idea is as follows: The operating system keeps several processes in memory simultaneously (Figure 1.12). The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another process. When *that* process needs to wait, the CPU switches to *another* process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If she has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast **response time**. Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or touch screen. Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Input, for example, may be

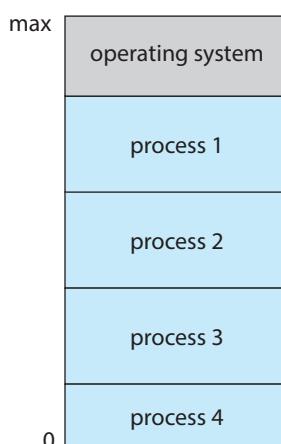


Figure 1.12 Memory layout for a multiprogramming system.

bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

Having several processes in memory at the same time requires some form of memory management, which we cover in Chapter 9 and Chapter 10. In addition, if several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is **CPU scheduling**, which is discussed in Chapter 5. Finally, running multiple processes concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text.

In a multitasking system, the operating system must ensure reasonable response time. A common method for doing so is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 10). The main advantage of this scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Multiprogramming and multitasking systems must also provide a file system (Chapter 13, Chapter 14, and Chapter 15). The file system resides on a secondary storage; hence, storage management must be provided (Chapter 11). In addition, a system must protect resources from inappropriate use (Chapter 17). To ensure orderly execution, the system must also provide mechanisms for process synchronization and communication (Chapter 6 and Chapter 7), and it may ensure that processes do not get stuck in a deadlock, forever waiting for one another (Chapter 8).

1.4.2 Dual-Mode and Multimode Operation

Since the operating system and its users share the hardware and software resources of the computer system, a properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs—or the operating system itself—to execute incorrectly. In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows differentiation among various modes of execution.

At the very least, we need two separate *modes* of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill

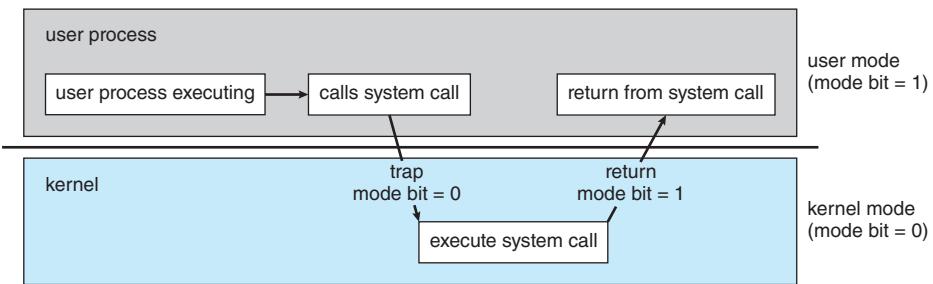


Figure 1.13 Transition from user to kernel mode.

the request. This is shown in Figure 1.13. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. Many additional privileged instructions are discussed throughout the text.

The concept of modes can be extended beyond two modes. For example, Intel processors have four separate **protection rings**, where ring 0 is kernel mode and ring 3 is user mode. (Although rings 1 and 2 could be used for various operating-system services, in practice they are rarely used.) ARMv8 systems have seven modes. CPUs that support virtualization (Section 18.1) frequently have a separate mode to indicate when the **virtual machine manager (VMM)** is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so.

We can now better understand the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call. Most contemporary operating systems—such as Microsoft Windows, Unix, and Linux—

take advantage of this dual-mode feature and provide greater protection for the operating system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems have a specific `syscall` instruction to invoke a system call.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space—then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

1.4.3 Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or

LINUX TIMERS

On Linux systems, the kernel configuration parameter HZ specifies the frequency of timer interrupts. An HZ value of 250 means that the timer generates 250 interrupts per second, or one interrupt every 4 milliseconds. The value of HZ depends upon how the kernel is configured, as well the machine type and architecture on which it is running. A related kernel variable is `jiffies`, which represent the number of timer interrupts that have occurred since the system was booted. A programming project in Chapter 2 further explores timing in the Linux kernel.

may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

1.5 Resource Management

As we have seen, an operating system is a **resource manager**. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

1.5.1 Process Management

A program can do nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A program such as a compiler is a process, and a word-processing program being run by an individual user on a PC is a process. Similarly, a social media app on a mobile device is a process. For now, you can consider a process to be an instance of a program in execution, but later you will see that the concept is more general. As described in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process running a web browser whose function is to display the contents of a web page on a screen. The process will be given the URL as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the screen. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process. A program is a *passive* entity, like the contents of a file stored on disk, whereas a process is an *active* entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads are covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although

two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU core—or in parallel across multiple CPU cores.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Scheduling processes and threads on the CPUs
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

We discuss process-management techniques in Chapter 3 through Chapter 7.

1.5.2 Memory Management

As discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The CPU reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). As noted earlier, the main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory-management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the *hardware* design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and which process is using them
- Allocating and deallocating memory space as needed
- Deciding which processes (or parts of processes) and data to move into and out of memory

Memory-management techniques are discussed in Chapter 9 and Chapter 10.

1.5.3 File-System Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. The operating system maps files onto physical media and accesses these files via the storage devices.

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Secondary storage is the most common, but tertiary storage is also possible. Each of these media has its own characteristics and physical organization. Most are controlled by a device, such as a disk drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields such as an mp3 music file). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass storage media and the devices that control them. In addition, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto mass storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques are discussed in Chapter 13, Chapter 14, and Chapter 15.

1.5.4 Mass-Storage Management

As we have already seen, the computer system must provide secondary storage to back up main memory. Most modern computer systems use HDDs and NVM devices as the principal on-line storage media for both programs and data. Most programs—including compilers, web browsers, word processors, and games—are stored on these devices until loaded into memory. The programs then use the devices as both the source and the destination of their processing. Hence, the proper management of secondary storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with secondary storage management:

- Mounting and unmounting
- Free-space management
- Storage allocation
- Disk scheduling
- Partitioning
- Protection

Because secondary storage is used frequently and extensively, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the secondary storage subsystem and the algorithms that manipulate that subsystem.

At the same time, there are many uses for storage that is slower and lower in cost (and sometimes higher in capacity) than secondary storage. Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD DVD and Blu-ray drives and platters are typical tertiary storage devices.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Techniques for secondary storage and tertiary storage management are discussed in Chapter 11.

1.5.5 Cache Management

Caching is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache.

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.14 Characteristics of various types of storage.

If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance, as you can see by examining Figure 1.14. Replacement algorithms for software-controlled caches are discussed in Chapter 10.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on hard disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the hard disk, in main memory, in the cache, and in an internal register (see Figure 1.15). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A

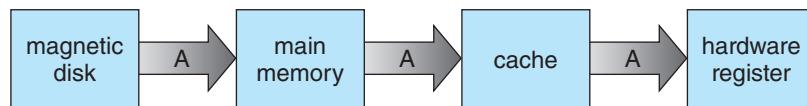


Figure 1.15 Migration of integer A from disk to register.

becomes the same only after the new value of A is written from the internal register back to the hard disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache (refer back to Figure 1.8). In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware issue (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 19.

1.5.6 I/O System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed earlier in this chapter how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 12, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

1.6 Security and Protection

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 17.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating-system security features are a fast-growing area of research and implementation. We discuss security in Chapter 16.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifier** (**user IDs**). In Windows parlance, this is a **security ID** (**SID**). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and **group identifier**. A user can be in one or more groups, depending on operating-system design

decisions. The user's group IDs are also included in every associated process and thread.

In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

1.7 Virtualization

Virtualization is a technology that allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. These environments can be viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other. A user of a **virtual machine** can switch among the various operating systems in the same way a user can switch among the various processes running concurrently in a single operating system.

Virtualization allows operating systems to run as applications within other operating systems. At first blush, there seems to be little reason for such functionality. But the virtualization industry is vast and growing, which is a testament to its utility and importance.

Broadly speaking, virtualization software is one member of a class that also includes emulation. **Emulation**, which involves simulating computer hardware in software, is typically used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU. That same concept can be extended to allow an entire operating system written for one platform to run on another. Emulation comes at a heavy price, however. Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code may run much more slowly than the native code.

With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently. Running multiple virtual machines allowed (and still allows) many users to run tasks on a system designed for a single user. Later, in response to problems with running multiple Microsoft Windows applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on Windows. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.16.)

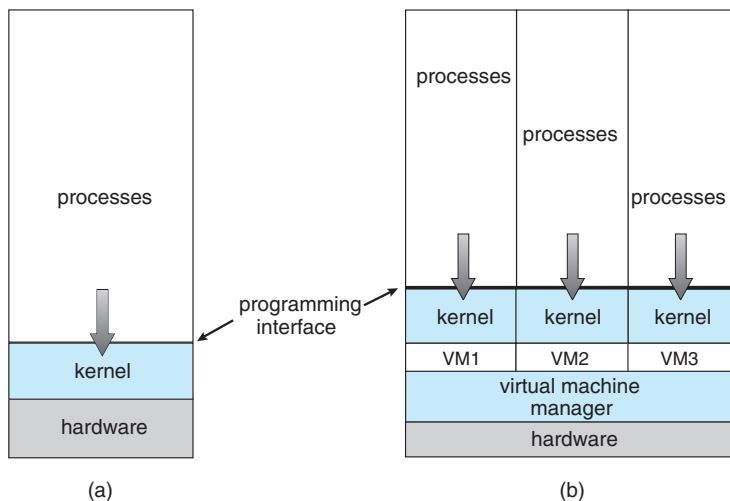


Figure 1.16 A computer running (a) a single operating system and (b) three virtual machines.

Windows was the **host** operating system, and the VMware application was the **virtual machine manager (VMM)**. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

Even though modern operating systems are fully capable of running multiple applications reliably, the use of virtualization continues to grow. On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host. For example, an Apple laptop running macOS on the x86 CPU can run a Windows 10 guest to allow execution of Windows applications. Companies writing software for multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware ESX and Citrix XenServer no longer run on host operating systems but rather *are* the host operating systems, providing services and resource management to virtual machine processes.

With this text, we provide a Linux virtual machine that allows you to run Linux—as well as the development tools we provide—on your personal system regardless of your host operating system. Full details of the features and implementation of virtualization can be found in Chapter 18.

1.8 Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver.

Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system’s utility and popularity.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. **TCP/IP** is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP, including all general-purpose ones. Some systems support proprietary protocols to suit their needs. For an operating system, it is necessary only that a network protocol have an interface device—a network adapter, for example—with a device driver to manage it, as well as software to handle data. These concepts are discussed throughout this book.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a building, or a campus. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network (MAN)** could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **personal-area network (PAN)** between a phone and a headset or a smartphone and a desktop computer.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate, they use or create a network. These networks also vary in their performance and reliability.

Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network. We cover computer networks and distributed systems in Chapter 19.

1.9 Kernel Data Structures

We turn next to a topic central to operating-system implementation: the way data are structured in the system. In this section, we briefly describe several fundamental data structures used extensively in operating systems. Readers

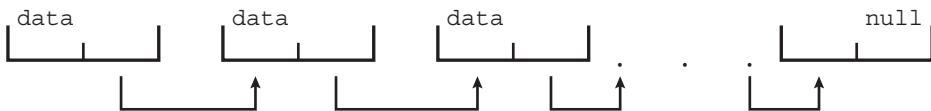


Figure 1.17 Singly linked list.

who require further details on these structures, as well as others, should consult the bibliography at the end of the chapter.

1.9.1 Lists, Stacks, and Queues

An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. If the data item being stored is larger than one byte, then multiple bytes can be allocated to the item, and the item is addressed as “item number \times item size.” But what about storing an item whose size may vary? And what about removing an item if the relative positions of the remaining items must be preserved? In such situations, arrays give way to other data structures.

After arrays, lists are perhaps the most fundamental data structures in computer science. Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence. The most common method for implementing this structure is a **linked list**, in which items are linked to one another. Linked lists are of several types:

- In a **singly linked list**, each item points to its successor, as illustrated in Figure 1.17.
- In a **doubly linked list**, a given item can refer either to its predecessor or to its successor, as illustrated in Figure 1.18.
- In a **circularly linked list**, the last element in the list refers to the first element, rather than to null, as illustrated in Figure 1.19.

Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items. One potential disadvantage of using a list is that performance for retrieving a specified item in a list of size n is linear— $O(n)$, as it requires potentially traversing all n elements in the worst case. Lists are sometimes used directly by kernel algorithms. Frequently, though, they are used for constructing more powerful data structures, such as stacks and queues.

A **stack** is a sequentially ordered data structure that uses the last in, first out (**LIFO**) principle for adding and removing items, meaning that the last item

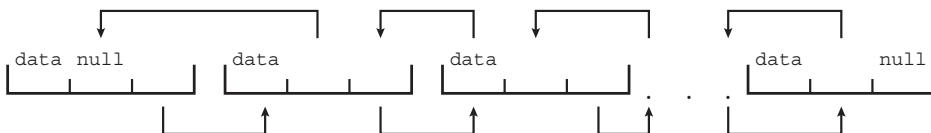


Figure 1.18 Doubly linked list.

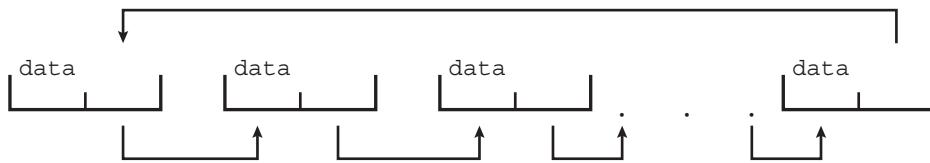


Figure 1.19 Circularly linked list.

placed onto a stack is the first item removed. The operations for inserting and removing items from a stack are known as *push* and *pop*, respectively. An operating system often uses a stack when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

A **queue**, in contrast, is a sequentially ordered data structure that uses the first in, first out (**FIFO**) principle: items are removed from a queue in the order in which they were inserted. There are many everyday examples of queues, including shoppers waiting in a checkout line at a store and cars waiting in line at a traffic signal. Queues are also quite common in operating systems—jobs that are sent to a printer are typically printed in the order in which they were submitted, for example. As we shall see in Chapter 5, tasks that are waiting to be run on an available CPU are often organized in queues.

1.9.2 Trees

A **tree** is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary tree**, a parent may have at most two children, which we term the *left child* and the *right child*. A **binary search tree** additionally requires an ordering between the parent’s two children in which *left_child* \leq *right_child*. Figure 1.20 provides an example of a binary search tree. When we search for an item in a binary search tree, the worst-case performance is $O(n)$ (consider how this can occur). To remedy this situation, we can use an algorithm to create a **balanced binary search tree**. Here, a tree containing n items has at most $\lg n$ levels, thus ensuring worst-case performance of $O(\lg n)$. We shall see in Section 5.7.1 that Linux uses a balanced binary search tree (known as a **red-black tree**) as part its CPU-scheduling algorithm.

1.9.3 Hash Functions and Maps

A **hash function** takes data as its input, performs a numeric operation on the data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Whereas searching for a data item through a list of size n can require up to $O(n)$ comparisons, using a hash function for retrieving data from a table can be as good as $O(1)$, depending on implementation details. Because of this performance, hash functions are used extensively in operating systems.

One potential difficulty with hash functions is that two unique inputs can result in the same output value—that is, they can link to the same table

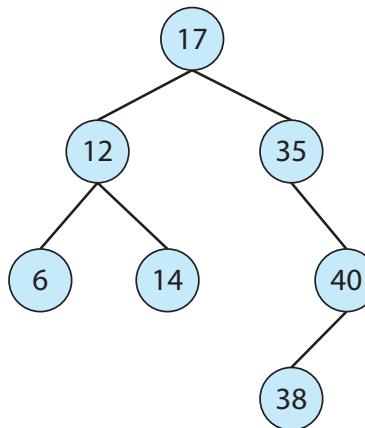


Figure 1.20 Binary search tree.

location. We can accommodate this *hash collision* by having a linked list at the table location that contains all of the items with the same hash value. Of course, the more collisions there are, the less efficient the hash function is.

One use of a hash function is to implement a **hash map**, which associates (or *maps*) [key:value] pairs using a hash function. Once the mapping is established, we can apply the hash function to the key to obtain the value from the hash map (Figure 1.21). For example, suppose that a user name is mapped to a password. Password authentication then proceeds as follows: a user enters her user name and password. The hash function is applied to the user name, which is then used to retrieve the password. The retrieved password is then compared with the password entered by the user for authentication.

1.9.4 Bitmaps

A **bitmap** is a string of n binary digits that can be used to represent the status of n items. For example, suppose we have several resources, and the availability of each resource is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice versa). The

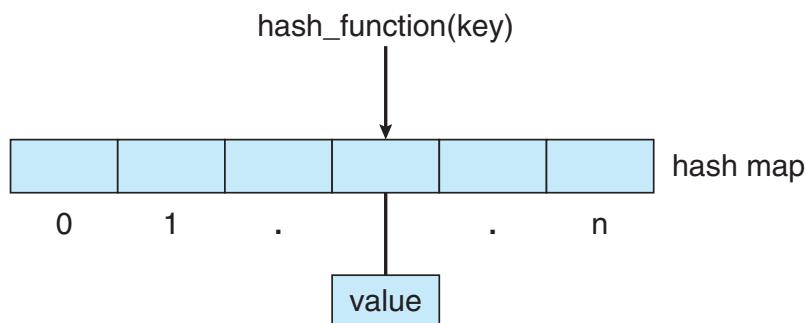


Figure 1.21 Hash map.

LINUX KERNEL DATA STRUCTURES

The data structures used in the Linux kernel are available in the kernel source code. The *include* file <linux/list.h> provides details of the linked-list data structure used throughout the kernel. A queue in Linux is known as a *kfifo*, and its implementation can be found in the *kfifo.c* file in the *kernel* directory of the source code. Linux also provides a balanced binary search tree implementation using *red-black trees*. Details can be found in the include file <linux/rbtree.h>.

value of the i^{th} position in the bitmap is associated with the i^{th} resource. As an example, consider the bitmap shown below:

001011101

Resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available.

The power of bitmaps becomes apparent when we consider their space efficiency. If we were to use an eight-bit Boolean value instead of a single bit, the resulting data structure would be eight times larger. Thus, bitmaps are commonly used when there is a need to represent the availability of a large number of resources. Disk drives provide a nice illustration. A medium-sized disk drive might be divided into several thousand individual units, called **disk blocks**. A bitmap can be used to indicate the availability of each disk block.

In summary, data structures are pervasive in operating system implementations. Thus, we will see the structures discussed here, along with others, throughout this text as we explore kernel algorithms and their implementations.

1.10 Computing Environments

So far, we have briefly described several aspects of computer systems and the operating systems that manage them. We turn now to a discussion of how operating systems are used in a variety of computing environments.

1.10.1 Traditional Computing

As computing has matured, the lines separating many of the traditional computing environments have blurred. Consider the “typical office environment.” Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers.

Today, web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish **portals**, which provide web accessibility to their internal servers. **Network computers** (or **thin clients**)—which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired. Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile devices can also

connect to **wireless networks** and cellular data networks to use the company's web portal (as well as the myriad other web resources).

At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive in many places, giving home users more access to more data. These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers. Many homes use **firewall** to protect their networks from security breaches. Firewalls limit the communications between devices on a network.

In the latter half of the 20th century, computing resources were relatively scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive. Batch systems processed jobs in bulk, with predetermined input from files or other data sources. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. These time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Traditional time-sharing systems are rare today. The same scheduling technique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time. Even a web browser can be composed of multiple processes, one for each website currently being visited, with time sharing applied to each web browser process.

1.10.2 Mobile Computing

Mobile computing refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing. Over the past few years, however, features on mobile devices have become so rich that the distinction in functionality between, say, a consumer laptop and a tablet computer may be difficult to discern. In fact, we might argue that the features of a contemporary mobile device allow it to provide functionality that is either unavailable or impractical on a desktop or laptop computer.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording and editing high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such devices. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on Earth. That functionality is

especially useful in designing applications that provide navigation—for example, telling users which way to walk or drive or perhaps directing them to nearby services, such as restaurants. An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in *augmented-reality* applications, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 256 GB in storage, it is not uncommon to find 8 TB in storage on a desktop computer. Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers. We examine these two mobile operating systems in further detail in Chapter 2.

1.10.3 Client–Server Computing

Contemporary network architecture features arrangements in which **server systems** satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure 1.22.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server

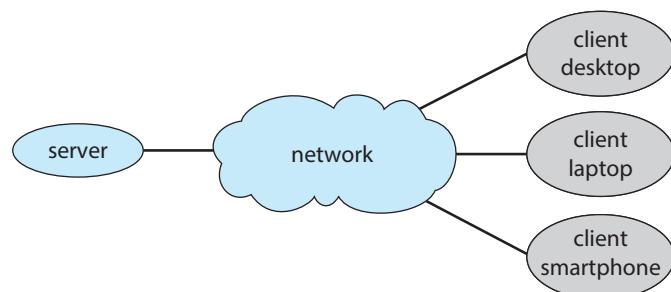


Figure 1.22 General structure of a client–server system.

running a database that responds to client requests for data is an example of such a system.

- The **file-serve system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers. The actual contents of the files can vary greatly, ranging from traditional web pages to rich multimedia content such as high-definition video.

1.10.4 Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client–server systems. In a client–server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network. Figure 1.23 illustrates such a scenario.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another. The Napster system used an approach similar to the first type described above: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used a technique similar to the second type: a client broadcast file requests to other nodes in the system, and nodes that could service the request responded directly to the client. Peer-to-peer networks can be used to exchange copyrighted materials (music, for example) anonymously, and there are laws governing the distribution of copyrighted material. Notably, Napster ran into legal trouble for copyright infringement, and its services were shut down in 2001. For this reason, the future of exchanging files remains uncertain.

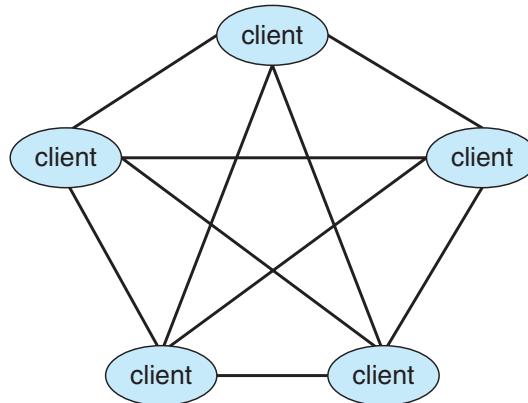


Figure 1.23 Peer-to-peer system with no centralized service.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP (VoIP)**. Skype uses a hybrid peer-to-peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

1.10.5 Cloud Computing

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud (**ec2**) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use. There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- Software as a service (**SaaS**)—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service (**PaaS**)—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service (**IaaS**)—servers or storage available over the Internet (for example, storage available for making backup copies of production data)

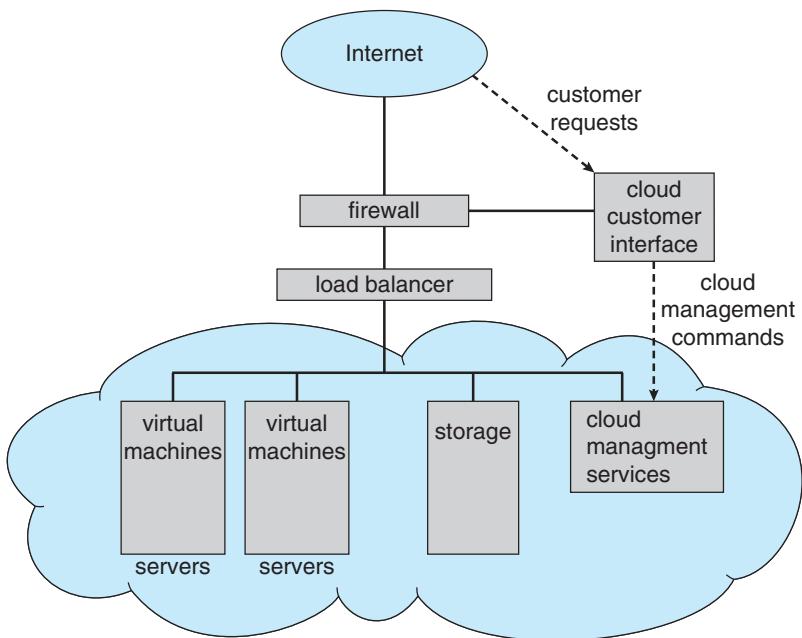


Figure 1.24 Cloud computing.

These cloud-computing types are not discrete, as a cloud computing environment may provide a combination of several types. For example, an organization may provide both SaaS and IaaS as publicly available services.

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs themselves are managed by cloud management tools, such as VMware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Figure 1.24 illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.

1.10.6 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to optical drives and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices

with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator will be able to notify the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing **must** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt *after* it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a traditional laptop system where it is desirable (but not mandatory) to respond quickly.

In Chapter 5, we consider the scheduling facility needed to implement real-time functionality in an operating system, and in Chapter 20 we describe the real-time components of Linux.

1.11 Free and Open-Source Operating Systems

The study of operating systems has been made easier by the availability of a vast number of free software and open-source releases. Both **free operating systems** and **open-source operating systems** are available in source-code format rather than as compiled binary code. Note, though, that free software and open-source software are two different ideas championed by different groups of people (see <http://gnu.org/philosophy/open-source-misses-the-point.html/> for a discussion on the topic). Free software (sometimes referred to as *frei/libre software*) not only makes source code available but also is licensed to allow no-cost use, redistribution, and modification. Open-source software does not necessarily offer such licensing. Thus, although all free software is open source, some open-source software is not “free.” GNU/Linux is the most famous open-source operating system, with some distributions free and others open source only (<http://www.gnu.org/distros/>). Microsoft Windows is a well-known example of the opposite **closed-source** approach. Windows is **proprietary** software—Microsoft owns it, restricts its use, and carefully protects its source code. Apple’s macOS operating system comprises a hybrid

approach. It contains an open-source kernel named Darwin but includes proprietary, closed-source components as well.

Starting with the source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite—**reverse engineering** the source code from the binaries—is quite a lot of work, and useful items such as comments are never recovered. Learning operating systems by examining the source code has other benefits as well. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool. This text includes projects that involve modifying operating-system source code, while also describing algorithms at a high level to be sure all important operating-system topics are covered. Throughout the text, we provide pointers to examples of open-source code for deeper study.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to write it, debug it, analyze it, provide support, and suggest changes. Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code. Companies that earn revenue from selling their programs often hesitate to open-source their code, but Red Hat and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

1.11.1 History

In the early days of modern computing (that is, the 1950s), software generally came with source code. The original hackers (computer enthusiasts) at MIT's Tech Model Railroad Club left their programs in drawers for others to work on. "Homebrew" user groups exchanged code during their meetings. Company-specific user groups, such as Digital Equipment Corporation's DECUS, accepted contributions of source-code programs, collected them onto tapes, and distributed the tapes to interested members. In 1970, Digital's operating systems were distributed as source code with no restrictions or copyright notice.

Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, as well as protecting their code and their ideas from their competitors. Although the Homebrew user groups of the 1970s exchanged code during their meetings, the operating systems for hobbyist machines (such as CPM) were proprietary. By 1980, proprietary software was the usual case.

1.11.2 Free Operating Systems

To counter the move to limit software use and redistribution, Richard Stallman in 1984 started developing a free, UNIX-compatible operating system called GNU(which is a recursive acronym for “GNU’s Not Unix!”). To Stallman, “free” refers to freedom of use, not price. The free-software movement does not object

to trading a copy for an amount of money but holds that users are entitled to four certain freedoms: (1) to freely run the program, (2) to study and change the source code, and to give or sell copies either (3) with or (4) without changes. In 1985, Stallman published the [GNU Manifesto](#), which argues that all software should be free. He also formed the [Free Software Foundation \(FSF\)](#) with the goal of encouraging the use and development of free software.

The FSF uses the copyrights on its programs to implement “copyleft,” a form of licensing invented by Stallman. Copylefting a work gives anyone that possesses a copy of the work the four essential freedoms that make the work free, with the condition that redistribution must preserve these freedoms. The [GNU General Public License \(GPL\)](#) is a common license under which free software is released. Fundamentally, the GPL requires that the source code be distributed with any binaries and that all copies (including modified versions) be released under the same GPL license. The Creative Commons “Attribution Sharealike” license is also a copyleft license; “sharealike” is another way of stating the idea of copyleft.

1.11.3 GNU/Linux

As an example of a free and open-source operating system, consider [GNU/Linux](#). By 1991, the GNU operating system was nearly complete. The GNU Project had developed compilers, editors, utilities, libraries, and games — whatever parts it could not find elsewhere. However, the GNU kernel never became ready for prime time. In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called “Linux” operating system to grow rapidly, enhanced by several thousand programmers. In 1991, Linux was not free software, as its license permitted only noncommercial redistribution. In 1992, however, Torvalds rereleased Linux under the GPL, making it free software (and also, to use a term coined later, “open source”).

The resulting GNU/Linux operating system (with the kernel properly called Linux but the full operating system including GNU tools called GNU/Linux) has spawned hundreds of unique [distributions](#), or custom builds, of the system. Major distributions include Red Hat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose. For example, Red Hat Enterprise Linux is geared to large commercial use. PCLinuxOS is a [live CD](#)—an operating system that can be booted and run from a CD-ROM without being installed on a system’s boot disk. A variant of PCLinuxOS—called PCLinuxOS Supergamer DVD—is a [live DVD](#) that includes graphics drivers and games. A gamer can run it on any compatible system simply by booting from the DVD. When the gamer is finished, a reboot of the system resets it to its installed operating system.

You can run Linux on a Windows (or other) system using the following simple, free approach:

1. Download the free Virtualbox VMM tool from

<https://www.virtualbox.org/>

and install it on your system.

2. Choose to install an operating system from scratch, based on an installation image like a CD, or choose pre-built operating-system images that can be installed and run more quickly from a site like

<http://virtualboxes.org/images/>

These images are preinstalled with operating systems and applications and include many flavors of GNU/Linux.

3. Boot the virtual machine within Virtualbox.

An alternative to using Virtualbox is to use the free program Qemu (<http://wiki.qemu.org/Download/>), which includes the `qemu-img` command for converting Virtualbox images to Qemu images to easily import them.

With this text, we provide a virtual machine image of GNU/Linux running the Ubuntu release. This image contains the GNU/Linux source code as well as tools for software development. We cover examples involving the GNU/Linux image throughout this text, as well as in a detailed case study in Chapter 20.

1.11.4 BSD UNIX

BSD UNIX has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley (UCB) came in source and binary form, but they were not open source because a license from AT&T was required. BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within Virtualbox, as described above for Linux. The source code comes with the distribution and is stored in `/usr/src/`. The kernel source code is in `/usr/src/sys`. For example, to examine the virtual memory implementation code in the FreeBSD kernel, see the files in `/usr/src/sys/vm`. Alternatively, you can simply view the source code online at <https://svnweb.freebsd.org>.

As with many open-source projects, this source code is contained in and controlled by a **version control system**—in this case, “subversion” (<https://subversion.apache.org/source-code>). Version control systems allow a user to “pull” an entire source code tree to his computer and “push” any changes back into the repository for others to then pull. These systems also provide other features, including an entire history of each file and a conflict resolution feature in case the same file is changed concurrently. Another

version control system is [git](#), which is used for GNU/Linux, as well as other programs (<http://www.git-scm.com>).

Darwin, the core kernel component of macOS, is based on BSD UNIX and is open-sourced as well. That source code is available from <http://www.opensource.apple.com/>. Every macOS release has its open-source components posted at that site. The name of the package that contains the kernel begins with “xnu.” Apple also provides extensive developer tools, documentation, and support at <http://developer.apple.com>.

THE STUDY OF OPERATING SYSTEMS

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/.

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (<http://www.vmware.com>) provides a free “player” for Windows on which hundreds of free “virtual appliances” can run. Virtualbox (<http://www.virtualbox.com>) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

In some cases, simulators of specific hardware are also available, allowing the operating system to run on “native” hardware, all within the confines of a modern computer and modern operating system. For example, a DECSYSTEM-20 simulator running on macOS can boot TOPS-20, load the source tapes, and modify and compile a new TOPS-20 kernel. An interested student can search the Internet to find the original papers that describe the operating system, as well as the original manuals.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Not so many years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

1.11.5 Solaris

Solaris is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. The purchase of Sun by Oracle in 2009, however, left the state of this project unclear.

Several groups interested in using OpenSolaris have expanded its features, and their working set is Project Illumos, which has expanded from the OpenSolaris base to include more features and to be the basis for several products. Illumos is available at <http://wiki.illumos.org>.

1.11.6 Open-Source Systems as Learning Tools

The free-software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like <http://freshmeat.net/> and <http://distrowatch.com/> provide portals to many of these projects. As we stated earlier, open-source projects enable students to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs. The availability of source code for historic projects, such as Multics, can help students to understand those projects and to build knowledge that will help in the implementation of new projects.

Another advantage of working with open-source operating systems is their diversity. GNU/Linux and BSD UNIX are both open-source operating systems, for instance, but each has its own goals, utility, licensing, and purpose. Sometimes, licenses are not mutually exclusive and cross-pollination occurs, allowing rapid improvements in operating-system projects. For example, several major components of OpenSolaris have been ported to BSD UNIX. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

1.12 Summary

- An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run.
- Interrupts are a key way in which hardware interacts with the operating system. A hardware device triggers an interrupt by sending a signal to the CPU to alert the CPU that some event requires attention. The interrupt is managed by the interrupt handler.
- For a computer to do its job of executing programs, the programs must be in main memory, which is the only large storage area that the processor can access directly.
- The main memory is usually a volatile storage device that loses its contents when power is turned off or lost.

- Nonvolatile storage is an extension of main memory and is capable of holding large quantities of data permanently.
- The most common nonvolatile storage device is a hard disk, which can provide storage of both programs and data.
- The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.
- Modern computer architectures are multiprocessor systems in which each CPU contains several computing cores.
- To best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute.
- Multitasking is an extension of multiprogramming wherein CPU scheduling algorithms rapidly switch between processes, providing users with a fast response time.
- To prevent user programs from interfering with the proper operation of the system, the system hardware has two modes: user mode and kernel mode.
- Various instructions are privileged and can be executed only in kernel mode. Examples include the instruction to switch to kernel mode, I/O control, timer management, and interrupt management.
- A process is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other.
- An operating system manages memory by keeping track of what parts of memory are being used and by whom. It is also responsible for dynamically allocating and freeing memory space.
- Storage space is managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.
- Operating systems provide mechanisms for protecting and securing the operating system and users. Protection measures control the access of processes or users to the resources made available by the computer system.
- Virtualization involves abstracting a computer's hardware into several different execution environments.
- Data structures that are used in an operating system include lists, stacks, queues, trees, and maps.
- Computing takes place in a variety of environments, including traditional computing, mobile computing, client-server systems, peer-to-peer systems, cloud computing, and real-time embedded systems.

- Free and open-source operating systems are available in source-code format. Free software is licensed to allow no-cost use, redistribution, and modification. GNU/Linux, FreeBSD, and Solaris are examples of popular open-source systems.

Practice Exercises

- 1.1 What are the three main purposes of an operating system?
- 1.2 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to “waste” resources? Why is such a system not really wasteful?
- 1.3 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?
- 1.4 Keeping in mind the various definitions of *operating system*, consider whether the operating system should include applications such as web browsers and mail programs. Argue both that it should and that it should not, and support your answers.
- 1.5 How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security)?
- 1.6 Which of the following instructions should be privileged?
 - a. Set value of timer.
 - b. Read the clock.
 - c. Clear memory.
 - d. Issue a trap instruction.
 - e. Turn off interrupts.
 - f. Modify entries in device-status table.
 - g. Switch from user to kernel mode.
 - h. Access I/O device.
- 1.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.
- 1.8 Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?
- 1.9 Timers could be used to compute the current time. Provide a short description of how this could be accomplished.
- 1.10 Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the

device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

- 1.11** Distinguish between the client–server and peer-to-peer models of distributed systems.

Further Reading

Many general textbooks cover operating systems, including [Stallings (2017)] and [Tanenbaum (2014)]. [Hennessy and Patterson (2012)] provide coverage of I/O systems and buses and of system architecture in general. [Kurose and Ross (2017)] provides a general overview of computer networks.

[Russinovich et al. (2017)] give an overview of Microsoft Windows and covers considerable technical detail about the system internals and components. [McDougall and Mauro (2007)] cover the internals of the Solaris operating system. The macOS and iOS internals are discussed in [Levin (2013)]. [Levin (2015)] covers the internals of Android. [Love (2010)] provides an overview of the Linux operating system and great detail about data structures used in the Linux kernel. The Free Software Foundation has published its philosophy at <http://www.gnu.org/philosophy/free-software-for-freedom.html>.

Bibliography

- [Hennessy and Patterson (2012)]** J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth Edition, Morgan Kaufmann (2012).
- [Kurose and Ross (2017)]** J. Kurose and K. Ross, *Computer Networking—A Top-Down Approach*, Seventh Edition, Addison-Wesley (2017).
- [Levin (2013)]** J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).
- [Levin (2015)]** J. Levin, *Android Internals—A Confectioner's Cookbook. Volume I* (2015).
- [Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).
- [McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).
- [Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).
- [Stallings (2017)]** W. Stallings, *Operating Systems, Internals and Design Principles (9th Edition)* Ninth Edition, Prentice Hall (2017).
- [Tanenbaum (2014)]** A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall (2014).

Chapter 1 Exercises

- 1.12 How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?
- 1.13 Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and disadvantages of each.
- 1.14 What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?
- 1.15 Explain how the Linux kernel variables `HZ` and `jiffies` can be used to determine the number of seconds the system has been running since it was booted.
- 1.16 Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.
 - a. How does the CPU interface with the device to coordinate the transfer?
 - b. How does the CPU know when the memory operations are complete?
 - c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.
- 1.17 Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.
- 1.18 Many SMP systems have different levels of caches; one level is local to each processing core, and another level is shared among all processing cores. Why are caching systems designed this way?
- 1.19 Rank the following storage systems from slowest to fastest:
 - a. Hard-disk drives
 - b. Registers
 - c. Optical disk
 - d. Main memory
 - e. Nonvolatile memory
 - f. Magnetic tapes
 - g. Cache

EX-2 Exercises

- 1.20** Consider an SMP system similar to the one shown in Figure 1.8. Illustrate with an example how data residing in memory could in fact have a different value in each of the local caches.
- 1.21** Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:
 - a. Single-processor systems
 - b. Multiprocessor systems
 - c. Distributed systems
- 1.22** Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.
- 1.23** Which network configuration—LAN or WAN—would best suit the following environments?
 - a. A campus student union
 - b. Several campus locations across a statewide university system
 - c. A neighborhood
- 1.24** Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs.
- 1.25** What are some advantages of peer-to-peer systems over client–server systems?
- 1.26** Describe some distributed applications that would be appropriate for a peer-to-peer system.
- 1.27** Identify several advantages and several disadvantages of open-source operating systems. Identify the types of people who would find each aspect to be an advantage or a disadvantage.

Operating-System Structures



An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating system designers. We consider what services an operating system provides, how they are provided, how they are debugged, and what the various methodologies are for designing such systems. Finally, we describe how operating systems are created and how a computer starts its operating system.

CHAPTER OBJECTIVES

- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Apply tools for monitoring operating system performance.
- Design and implement kernel modules for interacting with a Linux kernel.

2.1 Operating-System Services

An operating system provides an environment for the execution of programs. It makes certain services available to programs and to the users of those programs. The specific services provided, of course, differ from one operating

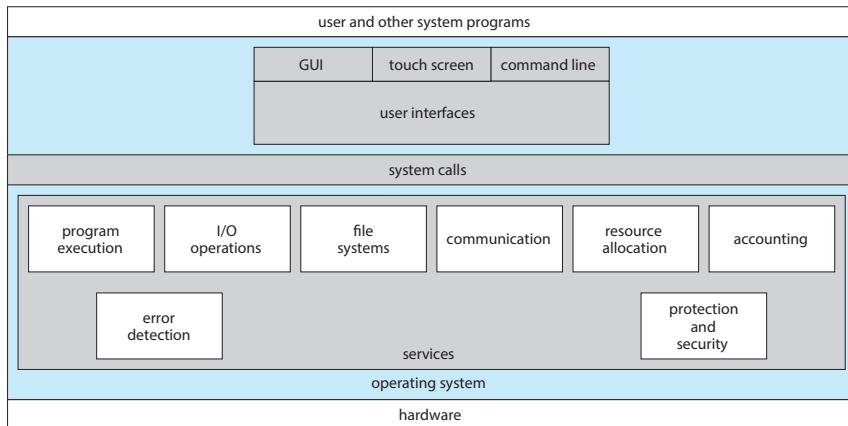


Figure 2.1 A view of operating system services.

system to another, but we can identify common classes. Figure 2.1 shows one view of the various operating-system services and how they interrelate. Note that these services also make the programming task easier for the programmer.

One set of operating system services provides functions that are helpful to the user.

- **User interface.** Almost all operating systems have a **user interface (UI)**. This interface can take several forms. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a **touch-screen interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Some systems provide two or all three of these variations.
- **Program execution.** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations.** A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation.** The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow

personal choice and sometimes to provide specific features or performance characteristics.

- **Communications.** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.
- **Error detection.** The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple processes can gain efficiency by sharing the computer resources among the different processes.

- **Resource allocation.** When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the process that must be executed, the number of processing cores on the CPU, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.
- **Logging.** We want to keep track of which programs use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for system administrators who wish to reconfigure the system to improve computing services.
- **Protection and security.** The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself

or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

2.2 User and Operating-System Interface

We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss three fundamental approaches. One provides a command-line interface, or **command interpreter**, that allows users to directly enter commands to be performed by the operating system. The other two allow users to interface with the operating system via a graphical user interface, or **GUI**.

2.2.1 Command Interpreters

Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a special program that is running when a process is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as **shells**. For example, on UNIX and Linux systems, a user may choose among several different shells, including the *C shell*, *Bourne-Again shell*, *Korn shell*, and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure 2.2 shows the *Bourne-Again* (or *bash*) shell command interpreter being used on macOS.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The various shells available on UNIX systems operate in this way. These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by UNIX, among other operating systems—implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

```
rm file.txt
```

would search for a file called `rm`, load the file into memory, and execute it with the parameter `file.txt`. The logic associated with the `rm` command would be

```

Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G  41% /
tmpfs          127G  520K  127G  1% /dev/shm
/dev/sda1       477M   71M   381M 16% /boot
/dev/dssd0000   1.0T  480G  545G 47% /dssd_xfs
tcp://192.168.150.1:3334/oranges
                  12T  5.7T  6.4T  47% /mnt/orangesfs
/dev/gpfs-test  23T  1.1T   22T  5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root     97653 11.2  6.6 42665344 17520636 ? S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root     69849  6.6  0.0    0  0 ? S Jul12 181:54 [vpthread-1-1]
root     69850  6.4  0.0    0  0 ? S Jul12 177:42 [vpthread-1-2]
root     3829  3.0  0.0    0  0 ? S Jun27 730:04 [rp_thread 7:0]
root     3826  3.0  0.0    0  0 ? S Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-rwx----- 1 root root 20667161 Jun  3 2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#

```

Figure 2.2 The bash shell command interpreter in macOS.

defined completely by the code in the file `rm`. In this way, programmers can add new commands to the system easily by creating new files with the proper program logic. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

2.2.2 Graphical User Interface

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window-and-menu system characterized by a desktop metaphor. The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer’s location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder—or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system has undergone various changes over the years, the most significant being the adoption of the *Aqua* interface that appeared with macOS. Microsoft’s first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system. Later versions of Windows have made significant changes in the appearance of the GUI along with several enhancements in its functionality.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however, with significant development in GUI designs from various open-source projects, such as *K Desktop Environment* (or *KDE*) and the *GNOME* desktop by the *GNU* project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.

2.2.3 Touch-Screen Interface

Because either a command-line interface or a mouse-and-keyboard system is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touch-screen interface. Here, users interact by making **gestures** on the touch screen—for example, pressing and swiping fingers across the screen. Although earlier smartphones included a physical keyboard, most smartphones and tablets now simulate a keyboard on the touch screen. Figure 2.3 illustrates the touch screen of the Apple iPhone. Both the iPad and the iPhone use the **Springboard** touch-screen interface.

2.2.4 Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the



Figure 2.3 The iPhone touch screen.

command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform. Indeed, on some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable. Further, command-line interfaces usually make repetitive tasks easier, in part because they have their own programmability. For example, if a frequent task requires a set of command-line steps, those steps can be recorded into a file, and that file can be run just like a program. The program is not compiled into executable code but rather is interpreted by the command-line interface. These shell scripts are very common on systems that are command-line oriented, such as UNIX and Linux.

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the shell interface. Recent versions of the Windows operating system provide both a standard GUI for desktop and traditional laptops and a touch screen for tablets. The various changes undergone by the Macintosh operating systems also provide a nice study in contrast. Historically, Mac OS has not provided a command-line interface, always requiring its users to interface with the operating system using its GUI. However, with the release of macOS (which is in part implemented using a UNIX kernel), the operating system now provides both an Aqua GUI and a command-line interface. Figure 2.4 is a screenshot of the macOS GUI.

Although there are apps that provide a command-line interface for iOS and Android mobile systems, they are rarely used. Instead, almost all users of mobile systems interact with their devices using the touch-screen interface.

The user interface can vary from system to system and even from user to user within a system; however, it typically is substantially removed from the actual system structure. The design of a useful and intuitive user interface is therefore not a direct function of the operating system. In this book, we concentrate on the fundamental problems of providing adequate service to

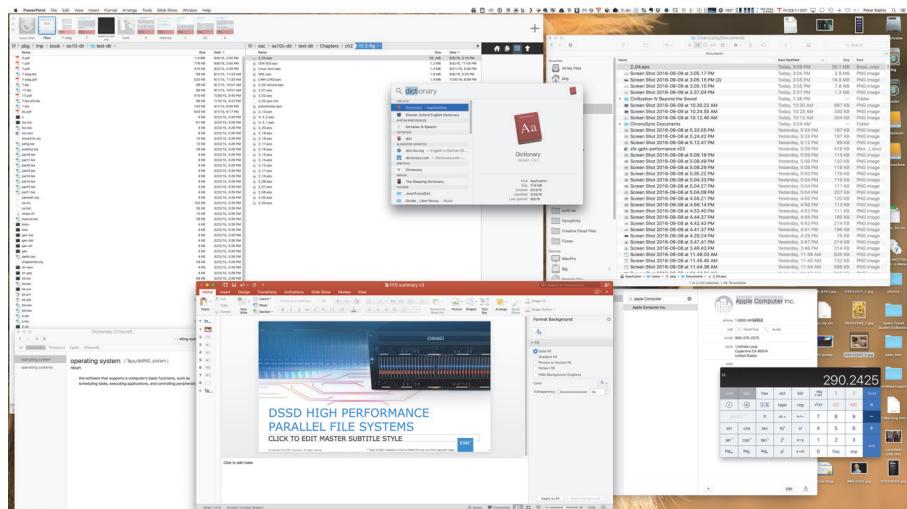


Figure 2.4 The macOS GUI.

user programs. From the point of view of the operating system, we do not distinguish between user programs and system programs.

2.3 System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

2.3.1 Example

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is to pass the names of the two files as part of the command—for example, the UNIX cp command:

```
cp in.txt out.txt
```

This command copies the input file `in.txt` to the output file `out.txt`. A second approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call. Possible error conditions for each system call must be handled. For example, when the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should output an error message (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been

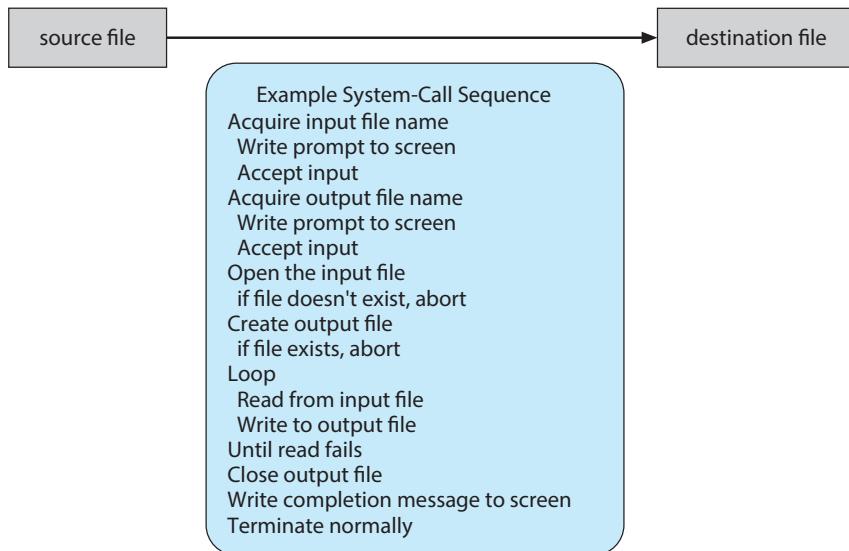


Figure 2.5 Example of how system calls are used.

reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (for example, no more available disk space).

Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in Figure 2.5.

2.3.2 Application Programming Interface

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and macOS), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**. Note that—unless specified—the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call.

Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function **CreateProcess()** (which, unsurprisingly, is used to create

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

<code>#include <unistd.h></code>		
<code>ssize_t</code>	<code>read(int fd, void *buf, size_t count)</code>	
<code>return value</code>	<code>function name</code>	<code>parameters</code>

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.

a new process) actually invokes the `NTCreateProcess()` system call in the Windows kernel.

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences often make this more difficult than it may appear). Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

Another important factor in handling system calls is the **run-time environment (RTE)**—the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders. The RTE provides a

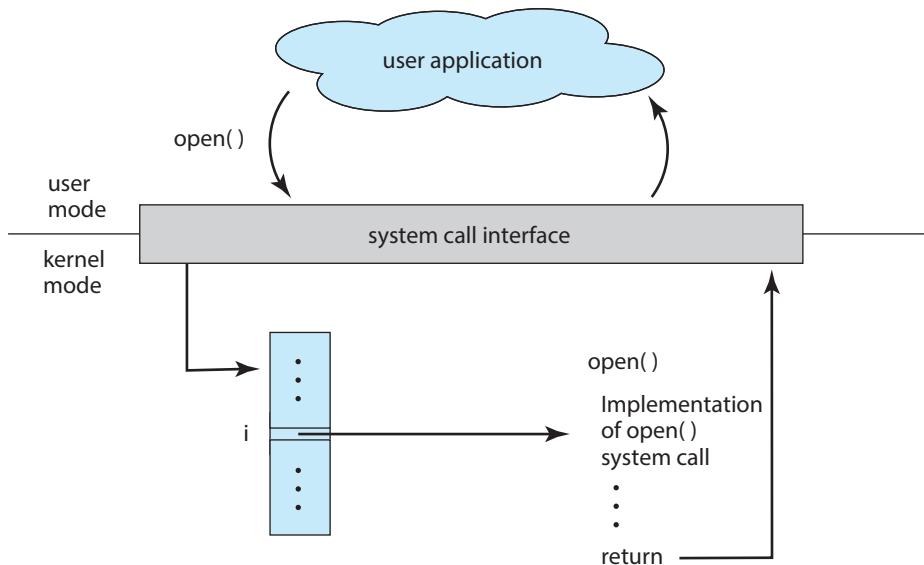


Figure 2.6 The handling of a user application invoking the `open()` system call.

system-call interface that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system-call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call.

The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the RTE. The relationship among an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the `open()` system call.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.7). Linux uses a combination of these approaches. If there are five or fewer parameters,

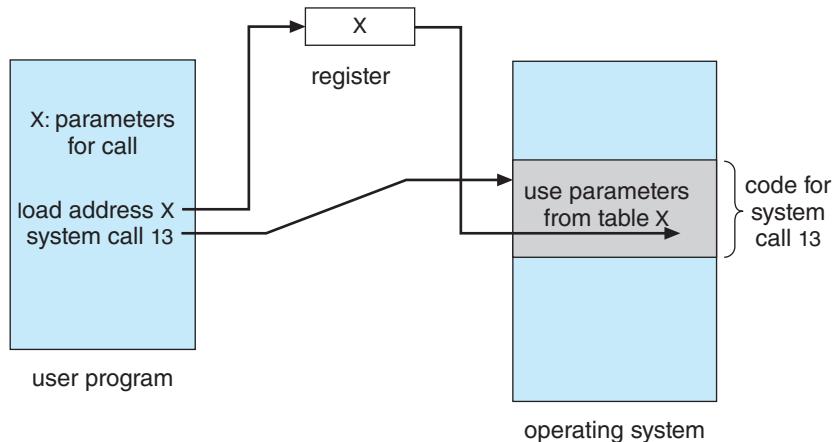


Figure 2.7 Passing of parameters as a table.

registers are used. If there are more than five parameters, the block method is used. Parameters also can be placed, or **pushed**, onto a **stack** by the program and **popped** off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

2.3.3 Types of System Calls

System calls can be grouped roughly into six major categories: **process control**, **file management**, **device management**, **information maintenance**, **communications**, and **protection**. Below, we briefly discuss the types of system calls that may be provided by an operating system. Most of these system calls support, or are supported by, concepts and functions that are discussed in later chapters. Figure 2.8 summarizes the types of system calls normally provided by an operating system. As mentioned, in this text, we normally refer to the system calls by generic names. Throughout the text, however, we provide examples of the actual counterparts to the system calls for UNIX, Linux, and Windows systems.

2.3.3.1 Process Control

A running program needs to be able to halt its execution either normally (`end()`) or abnormally (`abort()`). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to a special log file on disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to

- **Process control**
 - create process, terminate process
 - load, execute
 - get process attributes, set process attributes
 - wait event, signal event
 - allocate and free memory
 - **File management**
 - create file, delete file
 - open, close
 - read, write, reposition
 - get file attributes, set file attributes
 - **Device management**
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices
 - **Information maintenance**
 - get time or date, set time or date
 - get system data, set system data
 - get process, file, or device attributes
 - set process, file, or device attributes
 - **Communications**
 - create, delete communication connection
 - send, receive messages
 - transfer status information
 - attach or detach remote devices
 - **Protection**
 - get file permissions
 - set file permissions
-

Figure 2.8 Types of system calls.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

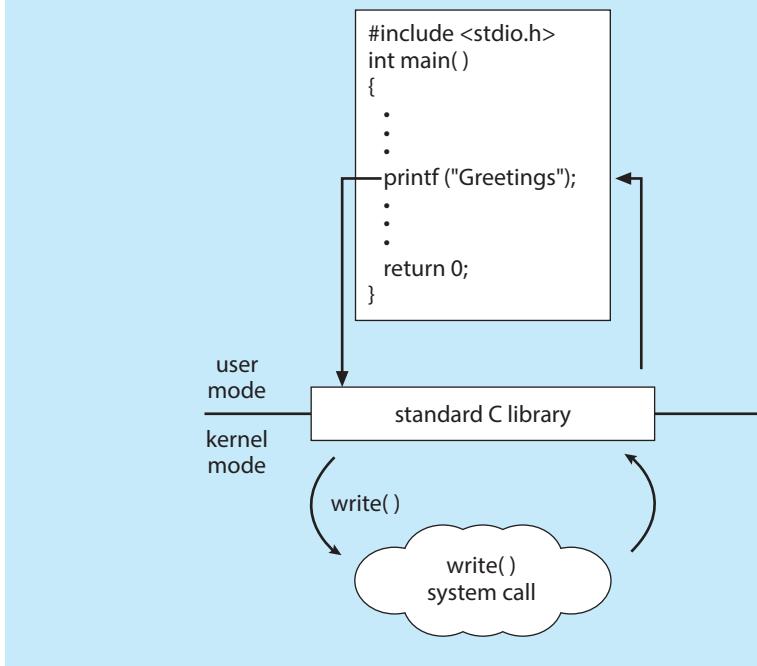
any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. Some systems may allow for special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. It is then possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

A process executing one program may want to load() and execute() another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command or the click of a mouse. An interesting question is where to return control when the loaded program terminates. This question is related to whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new process to be multiprogrammed. Often, there is a system call specifically for this purpose (`create_process()`).

If we create a new process, or perhaps even a set of processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a process, including the process's priority, its maximum allowable execution time, and so on (`get_process_attributes()` and `set_process_attributes()`). We may also want to terminate a process that we created (`terminate_process()`) if we find that it is incorrect or is no longer needed.

Having created new processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (`wait_time()`). More probably, we will want to wait for a specific event to occur (`wait_event()`). The processes should then signal when that event has occurred (`signal_event()`).

Quite often, two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing

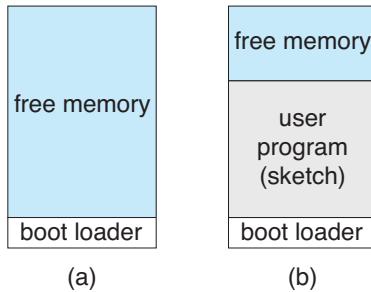


Figure 2.9 Arduino execution. (a) At system startup. (b) Running a sketch.

a process to **lock** shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include `acquire_lock()` and `release_lock()`. System calls of these types, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 6 and Chapter 7.

There are so many facets of and variations in process control that we next use two examples—one involving a single-tasking system and the other a multitasking system—to clarify these concepts. The Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, to just name a few. To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a **sketch**) from the PC to the Arduino's flash memory via a USB connection. The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a **boot loader** loads the sketch into a specific region in the Arduino's memory (Figure 2.9). Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to. For example, if the Arduino's temperature sensor detects that the temperature has exceeded a certain threshold, the sketch may have the Arduino start the motor for a fan. An Arduino is considered a single-tasking system, as only one sketch can be present in memory at a time; if another sketch is loaded, it replaces the existing sketch. Furthermore, the Arduino provides no user interface beyond hardware input sensors.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run, awaiting commands and running programs the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.10). To start a new process, the shell executes a `fork()` system call. Then, the selected program is loaded into memory via an `exec()` system call, and the program is executed. Depending on how the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately waits for another command to be entered. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an `exit()`

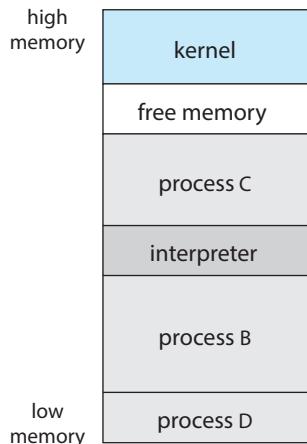


Figure 2.10 FreeBSD running multiple programs.

system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs. Processes are discussed in Chapter 3 with a program example using the `fork()` and `exec()` system calls.

2.3.3.2 File Management

The file system is discussed in more detail in Chapter 13 through Chapter 15. Here, we identify several common system calls dealing with files.

We first need to be able to `create()` and `delete()` files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to `open()` it and to use it. We may also `read()`, `write()`, or `reposition()` (rewind or skip to the end of the file, for example). Finally, we need to `close()` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to set them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, `get_file_attributes()` and `set_file_attributes()`, are required for this function. Some operating systems provide many more calls, such as calls for file `move()` and `copy()`. Others might provide an API that performs those operations using code and other system calls, and others might provide system programs to perform the tasks. If the system programs are callable by other programs, then each can be considered an API by other system programs.

2.3.3.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first `request()` a device, to ensure exclusive use of it. After we are finished with the device, we `release()` it. These functions are similar to the `open()` and `close()` system calls for files. Other operating systems allow unmanaged access to devices. The hazard then is the potential for device contention and perhaps deadlock, which are described in Chapter 8.

Once the device has been requested (and allocated to us), we can `read()`, `write()`, and (possibly) `reposition()` the device, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls is used on both files and devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar. This is another example of the many design decisions that go into building an operating system and user interface.

2.3.3.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current `time()` and `date()`. Other system calls may return information about the system, such as the version number of the operating system, the amount of free memory or disk space, and so on.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to `dump()` memory. This provision is useful for debugging. The program `strace`, which is available on Linux systems, lists each system call as it is executed. Even microprocessors provide a CPU mode, known as **single step**, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to get and set the process information (`get_process_attributes()` and `set_process_attributes()`). In Section 3.1.3, we discuss what information is normally kept.

2.3.3.5 Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to trans-

fer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a **host name** by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The `get_hostid()` and `get_processid()` system calls do this translation. The identifiers are then passed to the general-purpose `open()` and `close()` calls provided by the file system or to specific `open_connection()` and `close_connection()` system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept_connection()` call. Most processes that will be receiving connections are special-purpose **daemons**, which are system programs provided for that purpose. They execute a `wait_for_connection()` call and are awakened when a connection is made. The source of the communication, known as the **client**, and the receiving daemon, known as a **server**, then exchange messages by using `read_message()` and `write_message()` system calls. The `close_connection()` call terminates the communication.

In the **shared-memory model**, processes use `shared_memory_create()` and `shared_memory_attach()` system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Such mechanisms are discussed in Chapter 6. In Chapter 4, we look at a variation of the process scheme—threads—in which some memory is shared by default.

Both of the models just discussed are common in operating systems, and most systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

2.3.3.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection.

Typically, system calls providing protection include `set_permission()` and `get_permission()`, which manipulate the permission settings of

resources such as files and disks. The `allow_user()` and `deny_user()` system calls specify whether particular users can—or cannot—be allowed access to certain resources. We cover protection in Chapter 17 and the much larger issue of security—which involves using protection against external threats—in Chapter 16.

2.4 System Services

Another aspect of a modern system is its collection of system services. Recall Figure 1.1, which depicted the logical computer hierarchy. At the lowest level is hardware. Next is the operating system, then the system services, and finally the application programs. **System services**, also known as **system utilities**, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

- **File management.** These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.
- **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support.** Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.
- **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.
- **Background services.** All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to

run until the system is halted. Constantly running system-program processes are known as **services**, **subsystems**, or daemons. One example is the network daemon discussed in Section 2.3.3.5. In that example, a system needed a service to listen for network connections in order to connect those requests to the correct processes. Other examples include process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens of daemons. In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such **application programs** include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider a user's PC. When a user's computer is running the macOS operating system, the user might see the GUI, featuring a mouse-and-windows interface. Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways. Further confusing the user view, consider the user dual-booting from macOS into Windows. Now the same user on the same hardware has two entirely different interfaces and two sets of applications using the same physical resources. On the same hardware, then, a user can be exposed to multiple user interfaces sequentially or concurrently.

2.5 Linkers and Loaders

Usually, a program resides on disk as a binary executable file—for example, `a.out` or `prog.exe`. To run on a CPU, the program must be brought into memory and placed in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core. The steps are highlighted in Figure 2.11.

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an **relocatable object file**. Next, the **linker** combines these relocatable object files into a single binary **executable** file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specified with the flag `-lm`).

A **loader** is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is **relocation**, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that, for example, the code can call library functions and access its variables as it executes. In Figure 2.11, we see that to run the loader, all that is necessary is to enter the name of the executable file on the command line. When a program name is entered on the

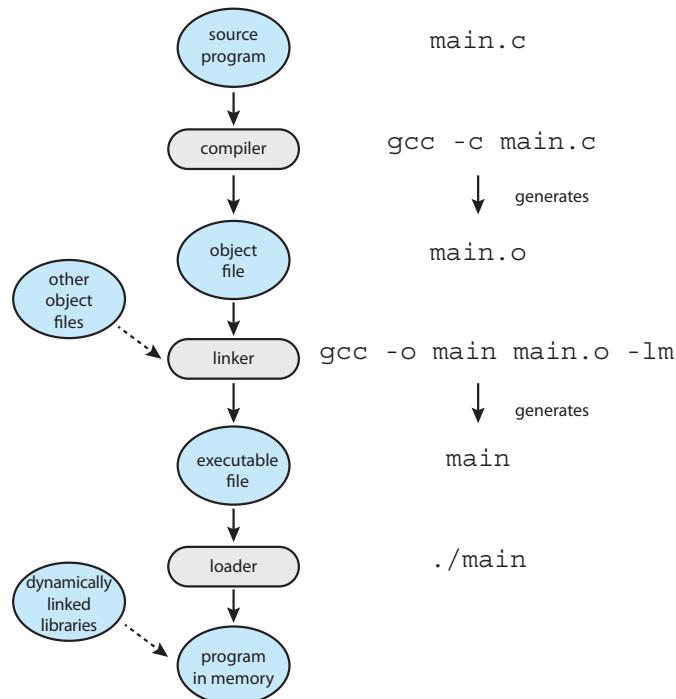


Figure 2.11 The role of the linker and loader.

command line on UNIX systems—for example, `./main`—the shell first creates a new process to run the program using the `fork()` system call. The shell then invokes the loader with the `exec()` system call, passing `exec()` the name of the executable file. The loader then loads the specified program into memory using the address space of the newly created process. (When a GUI interface is used, double-clicking on the icon associated with the executable file invokes the loader using a similar mechanism.)

The process described thus far assumes that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to dynamically link libraries as the program is loaded. Windows, for instance, supports dynamically linked libraries (**DLLs**). The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file. Instead, the library is conditionally linked and is loaded if it is required during program run time. For example, in Figure 2.11, the math library is not linked into the executable file `main`. Rather, the linker inserts relocation information that allows it to be dynamically linked and loaded as the program is loaded. We shall see in Chapter 9 that it is possible for multiple processes to share dynamically linked libraries, resulting in a significant savings in memory use.

Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program. For UNIX and Linux systems, this standard format is known as ELF (for **Executable and Linkable Format**). There are separate ELF formats for relocatable and

ELF FORMAT

Linux provides various commands to identify and evaluate ELF files. For example, the `file` command determines a file type. If `main.o` is an object file, and `main` is an executable file, the command

```
file main.o
```

will report that `main.o` is an ELF relocatable file, while the command

```
file main
```

will report that `main` is an ELF executable. ELF files are divided into a number of sections and can be evaluated using the `readelf` command.

executable files. One piece of information in the ELF file for executable files is the program's *entry point*, which contains the address of the first instruction to be executed when the program runs. Windows systems use the **Portable Executable** (PE) format, and macOS uses the **Mach-O** format.

2.6 Why Applications Are Operating-System Specific

Fundamentally, applications compiled on one operating system are not executable on other operating systems. If they were, the world would be a better place, and our choice of what operating system to use would depend on utility and features rather than which applications were available.

Based on our earlier discussion, we can now see part of the problem—each operating system provides a unique set of system calls. System calls are part of the set of services provided by operating systems for use by applications. Even if system calls were somehow uniform, other barriers would make it difficult for us to execute application programs on different operating systems. But if you have used multiple operating systems, you may have used some of the same applications on them. How is that possible?

An application can be made available to run on multiple operating systems in one of three ways:

1. The application can be written in an interpreted language (such as Python or Ruby) that has an interpreter available for multiple operating systems. The interpreter reads each line of the source program, executes equivalent instructions on the native instruction set, and calls native operating system calls. Performance suffers relative to that for native applications, and the interpreter provides only a subset of each operating system's features, possibly limiting the feature sets of the associated applications.
2. The application can be written in a language that includes a virtual machine containing the running application. The virtual machine is part of the language's full RTE. One example of this method is Java. Java has an RTE that includes a loader, byte-code verifier, and other components that load the Java application into the Java virtual machine. This RTE has been

ported, or developed, for many operating systems, from mainframes to smartphones, and in theory any Java app can run within the RTE wherever it is available. Systems of this kind have disadvantages similar to those of interpreters, discussed above.

3. The application developer can use a standard language or API in which the compiler generates binaries in a machine- and operating-system-specific language. The application must be ported to each operating system on which it will run. This porting can be quite time consuming and must be done for each new version of the application, with subsequent testing and debugging. Perhaps the best-known example is the POSIX API and its set of standards for maintaining source-code compatibility between different variants of UNIX-like operating systems.

In theory, these three approaches seemingly provide simple solutions for developing applications that can run across different operating systems. However, the general lack of application mobility has several causes, all of which still make developing cross-platform applications a challenging task. At the application level, the libraries provided with the operating system contain APIs to provide features like GUI interfaces, and an application designed to call one set of APIs (say, those available from iOS on the Apple iPhone) will not work on an operating system that does not provide those APIs (such as Android). Other challenges exist at lower levels in the system, including the following.

- Each operating system has a binary format for applications that dictates the layout of the header, instructions, and variables. Those components need to be at certain locations in specified structures within an executable file so the operating system can open the file and load the application for proper execution.
- CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly.
- Operating systems provide system calls that allow applications to request various activities, such as creating files and opening network connections. Those system calls vary among operating systems in many respects, including the specific operands and operand ordering used, how an application invokes the system calls, their numbering and number, their meanings, and their return of results.

There are some approaches that have helped address, though not completely solve, these architectural differences. For example, Linux—and almost every UNIX system—has adopted the ELF format for binary executable files. Although ELF provides a common standard across Linux and UNIX systems, the ELF format is not tied to any specific computer architecture, so it does not guarantee that an executable file will run across different hardware platforms.

APIs, as mentioned above, specify certain functions at the application level. At the architecture level, an **application binary interface** (ABI) is used to define how different components of binary code can interface for a given operating system on a given architecture. An ABI specifies low-level details, including address width, methods of passing parameters to system calls, the organization

of the run-time stack, the binary format of system libraries, and the size of data types, just to name a few. Typically, an ABI is specified for a given architecture (for example, there is an ABI for the ARMv8 processor). Thus, an ABI is the architecture-level equivalent of an API. If a binary executable file has been compiled and linked according to a particular ABI, it should be able to run on different systems that support that ABI. However, because a particular ABI is defined for a certain operating system running on a given architecture, ABIs do little to provide cross-platform compatibility.

In sum, all of these differences mean that unless an interpreter, RTE, or binary executable file is written for and compiled on a specific operating system on a specific CPU type (such as Intel x86 or ARMv8), the application will fail to run. Imagine the amount of work that is required for a program such as the Firefox browser to run on Windows, macOS, various Linux releases, iOS, and Android, sometimes on various CPU architectures.

2.7 Operating-System Design and Implementation

In this section, we discuss problems we face in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

2.7.1 Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: traditional desktop/laptop, mobile, distributed, or real time.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: **user goals** and **system goals**.

Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

A similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system. The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for Wind River VxWorks, a real-time operating system for embedded systems, must have been substantially different from those for Windows Server, a large multiaccess operating system designed for enterprise applications.

Specifying and designing an operating system is a highly creative task. Although no textbook can tell you how to do it, general principles have been

developed in the field of **software engineering**, and we turn now to a discussion of some of these principles.

2.7.2 Mechanisms and Policies

One important principle is the separation of **policy** from **mechanism**. Mechanisms determine *how* to do something; policies determine *what* will be done. For example, the timer construct (see Section 1.4.3) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism flexible enough to work across a range of policies is preferable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Microkernel-based operating systems (discussed in Section 2.8.3) take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or user programs themselves. In contrast, consider Windows, an enormously popular commercial operating system available for over three decades. Microsoft has closely encoded both mechanism and policy into the system to enforce a global look and feel across all devices that run the Windows operating system. All applications have similar interfaces, because the interface itself is built into the kernel and system libraries. Apple has adopted a similar strategy with its macOS and iOS operating systems.

We can make a similar comparison between commercial and open-source operating systems. For instance, contrast Windows, discussed above, with Linux, an open-source operating system that runs on a wide range of computing devices and has been available for over 25 years. The “standard” Linux kernel has a specific CPU scheduling algorithm (covered in Section 5.7.1), which is a mechanism that supports a certain policy. However, anyone is free to modify or replace the scheduler to support a different policy.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is *how* rather than *what*, it is a mechanism that must be determined.

2.7.3 Implementation

Once an operating system is designed, it must be implemented. Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

Early operating systems were written in assembly language. Now, most are written in higher-level languages such as C or C++, with small amounts

of the system written in assembly language. In fact, more than one higher-level language is often used. The lowest levels of the kernel might be written in assembly language and C. Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages. Android provides a nice example: its kernel is written mostly in C with some assembly language. Most Android system libraries are written in C or C++, and its application frameworks—which provide the developer interface to the system—are written mostly in Java. We cover Android’s architecture in more detail in Section 2.8.5.2.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those gained when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to port to other hardware if it is written in a higher-level language. This is particularly important for operating systems that are intended to run on several different hardware systems, such as small embedded devices, Intel x86 systems, and ARM chips running on phones and tablets.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is not a major issue in today’s systems. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind.

As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code. In addition, although operating systems are large, only a small amount of the code is critical to high performance; the interrupt handlers, I/O manager, memory manager, and CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottlenecks can be identified and can be refactored to operate more efficiently.

2.8 Operating-System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one single system. Each of these modules should be a well-defined portion of the system, with carefully defined interfaces and functions. You may use a similar approach when you structure your programs: rather than placing all of your code in the `main()` function, you instead separate logic into a number of functions, clearly articulate parameters and return values, and then call those functions from `main()`.

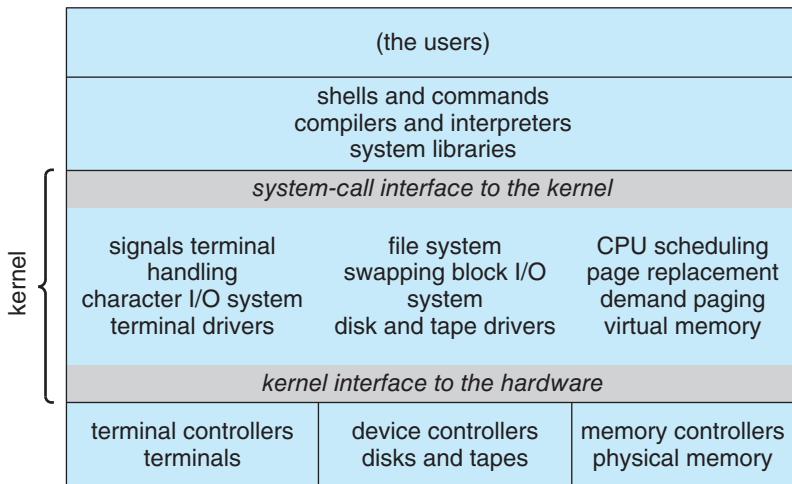


Figure 2.12 Traditional UNIX system structure.

We briefly discussed the common components of operating systems in Chapter 1. In this section, we discuss how these components are interconnected and melded into a kernel.

2.8.1 Monolithic Structure

The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a **monolithic** structure—is a common technique for designing operating systems.

An example of such limited structuring is the original UNIX operating system, which consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

The Linux operating system is based on UNIX and is structured similarly, as shown in Figure 2.13. Applications typically use the `glibc` standard C library when communicating with the system call interface to the kernel. The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but as we shall see in Section 2.8.4, it does have a modular design that allows the kernel to be modified during run time.

Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead in the system-call interface, and communication within the kernel is fast. Therefore, despite the drawbacks

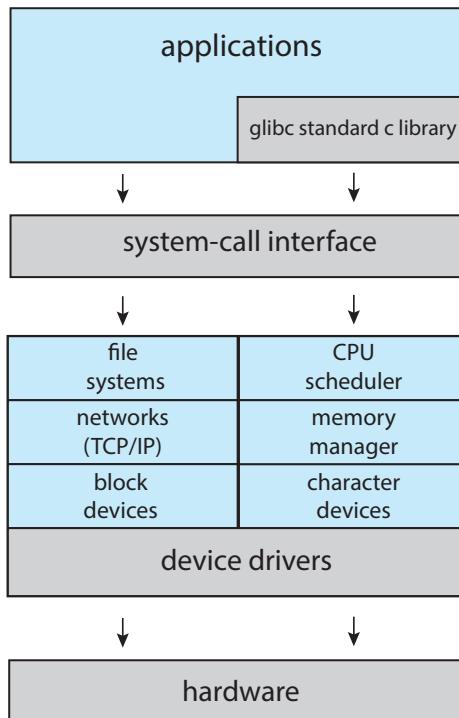


Figure 2.13 Linux system structure.

of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

2.8.2 Layered Approach

The monolithic approach is often known as a **tightly coupled** system because changes to one part of the system can have wide-ranging effects on other parts. Alternatively, we could design a **loosely coupled** system. Such a system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure 2.14.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer M—consists of data structures and a set of functions that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations)

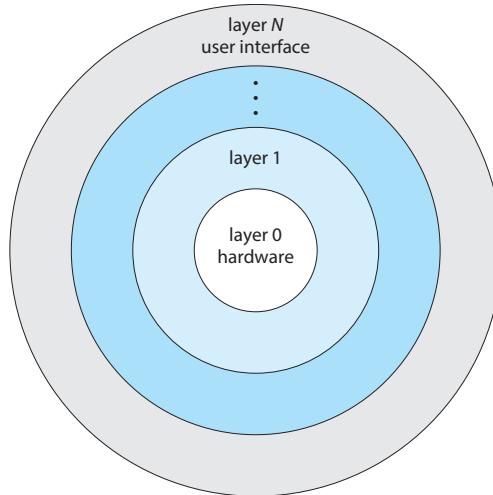


Figure 2.14 A layered operating system.

and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.

Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

Layered systems have been successfully used in computer networks (such as TCP/IP) and web applications. Nevertheless, relatively few operating systems use a pure layered approach. One reason involves the challenges of appropriately defining the functionality of each layer. In addition, the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service. *Some* layering is common in contemporary operating systems, however. Generally, these systems have fewer layers with more functionality, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

2.8.3 Microkernels

We have already seen that the original UNIX system had a monolithic structure. As UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing

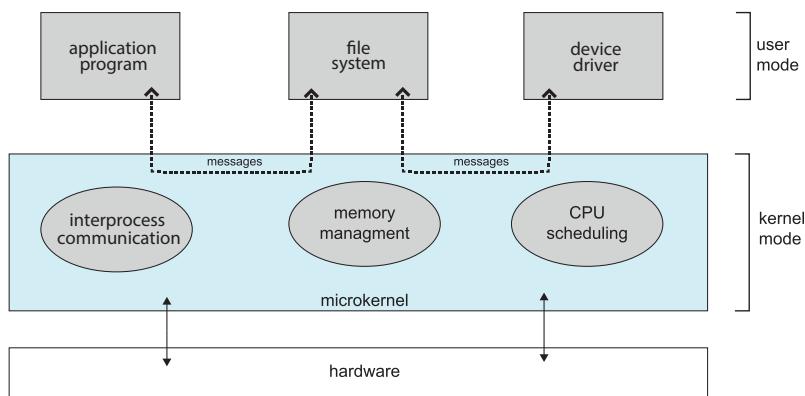


Figure 2.15 Architecture of a typical microkernel.

all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.15 illustrates the architecture of a typical microkernel.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through message passing, which was described in Section 2.3.3.5. For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Perhaps the best-known illustration of a microkernel operating system is *Darwin*, the kernel component of the macOS and iOS operating systems. Darwin, in fact, consists of two kernels, one of which is the Mach microkernel. We will cover the macOS and iOS systems in further detail in Section 2.8.5.1.

Another example is QNX, a real-time operating system for embedded systems. The QNX Neutrino microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Unfortunately, the performance of microkernels can suffer due to increased system-function overhead. When two user-level services must communicate, messages must be copied between the services, which reside in separate

address spaces. In addition, the operating system may have to switch from one process to the next to exchange the messages. The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems. Consider the history of Windows NT: The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95. Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel. Section 2.8.5.1 will describe how macOS addresses the performance issues of the Mach microkernel.

2.8.4 Modules

Perhaps the best current methodology for operating-system design involves using **loadable kernel modules (LKMs)**. Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.

The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be “inserted” into the kernel as the system is started (or *booted*) or during run time, such as when a USB device is plugged into a running machine. If the Linux kernel does not have the necessary driver, it can be dynamically loaded. LKMs can be removed from the kernel during run time as well. For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system. We cover creating LKMs in Linux in several programming exercises at the end of this chapter.

2.8.5 Hybrid Systems

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance. However, it is also modular, so that new functionality can be dynamically added to the kernel. Windows is largely

monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system *personalities*) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules. We provide case studies of Linux and Windows 10 in Chapter 20 and Chapter 21, respectively. In the remainder of this section, we explore the structure of three hybrid systems: the Apple macOS operating system and the two most prominent mobile operating systems—iOS and Android.

2.8.5.1 macOS and iOS

Apple's macOS operating system is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer. Architecturally, macOS and iOS have much in common, and so we present them together, highlighting what they share as well as how they differ from each other. The general architecture of these two systems is shown in Figure 2.16. Highlights of the various layers include the following:

- **User experience layer.** This layer defines the software interface that allows users to interact with the computing devices. macOS uses the *Aqua* user interface, which is designed for a mouse or trackpad, whereas iOS uses the *Springboard* user interface, which is designed for touch devices.
- **Application frameworks layer.** This layer includes the *Cocoa* and *Cocoa Touch* frameworks, which provide an API for the Objective-C and Swift programming languages. The primary difference between Cocoa and Cocoa Touch is that the former is used for developing macOS applications, and the latter by iOS to provide support for hardware features unique to mobile devices, such as touch screens.
- **Core frameworks.** This layer defines frameworks that support graphics and media including, Quicktime and OpenGL.

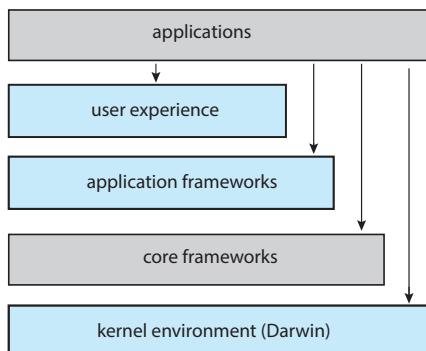


Figure 2.16 Architecture of Apple's macOS and iOS operating systems.

- **Kernel environment.** This environment, also known as **Darwin**, includes the Mach microkernel and the BSD UNIX kernel. We will elaborate on Darwin shortly.

As shown in Figure 2.16, applications can be designed to take advantage of user-experience features or to bypass them and interact directly with either the application framework or the core framework. Additionally, an application can forego frameworks entirely and communicate directly with the kernel environment. (An example of this latter situation is a C program written with no user interface that makes POSIX system calls.)

Some significant distinctions between macOS and iOS include the following:

- Because macOS is intended for desktop and laptop computer systems, it is compiled to run on Intel architectures. iOS is designed for mobile devices and thus is compiled for ARM-based architectures. Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.
- The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers. For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

We now focus on Darwin, which uses a hybrid structure. Darwin is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel. Darwin's structure is shown in Figure 2.17.

Whereas most operating systems provide a single system-call interface to the kernel—such as through the standard C library on UNIX and Linux systems—Darwin provides *two* system-call interfaces: Mach system calls (known as

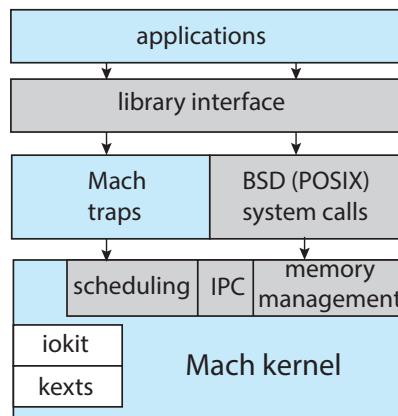


Figure 2.17 The structure of Darwin.

traps) and BSD system calls (which provide POSIX functionality). The interface to these system calls is a rich set of libraries that includes not only the standard C library but also libraries that provide networking, security, and programming language support (to name just a few).

Beneath the system-call interface, Mach provides fundamental operating-system services, including memory management, CPU scheduling, and inter-process communication (IPC) facilities such as message passing and remote procedure calls (RPCs). Much of the functionality provided by Mach is available through **kernel abstractions**, which include tasks (a Mach process), threads, memory objects, and ports (used for IPC). As an example, an application may create a new process using the BSD POSIX `fork()` system call. Mach will, in turn, use a task kernel abstraction to represent the process in the kernel.

In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which macOS refers to as **kernel extensions**, or **kexts**).

In Section 2.8.3, we described how the overhead of message passing between different services running in user space compromises the performance of microkernels. To address such performance problems, Darwin combines Mach, BSD, the I/O kit, and any kernel extensions into a single address space. Thus, Mach is not a pure microkernel in the sense that various subsystems run in user space. Message passing within Mach still does occur, but no copying is necessary, as the services have access to the same address space.

Apple has released the Darwin operating system as open source. As a result, various projects have added extra functionality to Darwin, such as the X-11 windowing system and support for additional file systems. Unlike Darwin, however, the Cocoa interface, as well as other proprietary Apple frameworks available for developing macOS applications, are closed.

2.8.5.2 Android

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity. The structure of Android appears in Figure 2.18.

Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features. These features, in turn, provide a platform for developing mobile applications that run on a multitude of Android-enabled devices.

Software designers for Android devices develop applications in the Java language, but they do not generally use the standard Java API. Google has designed a separate Android API for Java development. Java applications are compiled into a form that can execute on the Android RunTime ART, a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities. Java programs are first compiled to a Java bytecode .class file and then translated into an executable .dex file. Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs **ahead-of-time (AOT)** compil-

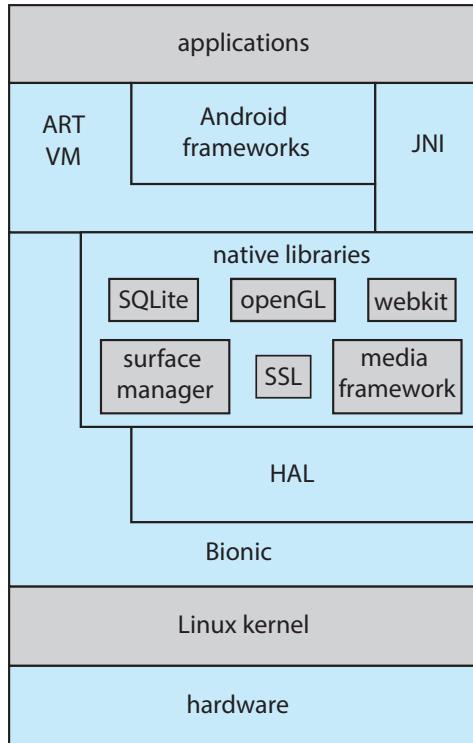


Figure 2.18 Architecture of Google’s Android.

tion. Here, .dex files are compiled into native machine code when they are installed on a device, from which they can execute on the ART. AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.

Android developers can also write Java programs that use the Java native interface—or JNI—which allows developers to bypass the virtual machine and instead write Java programs that can access specific hardware features. Programs written using JNI are generally not portable from one hardware device to another.

The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).

Because Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the hardware abstraction layer, or HAL. By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of specific hardware. This feature, of course, allows developers to write programs that are portable across different hardware platforms.

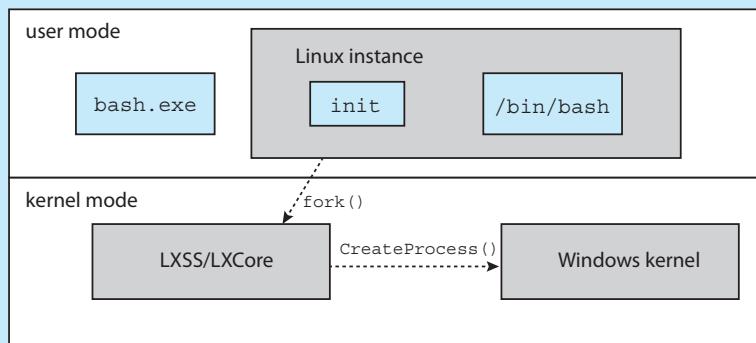
The standard C library used by Linux systems is the GNU C library (glibc). Google instead developed the **Bionic** standard C library for Android. Not only does Bionic have a smaller memory footprint than glibc, but it also has been designed for the slower CPUs that characterize mobile devices. (In addition, Bionic allows Google to bypass GPL licensing of glibc.)

At the bottom of Android's software stack is the Linux kernel. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management. It has also made changes in memory management and allocation and has added a new form of IPC known as *Binder* (which we will cover in Section 3.8.2.1).

WINDOWS SUBSYSTEM FOR LINUX

Windows uses a hybrid architecture that provides subsystems to emulate different operating-system environments. These user-mode subsystems communicate with the Windows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux ([WSL](#)), which allows native Linux applications (specified as ELF binaries) to run on Windows 10. The typical operation is for a user to start the Windows application `bash.exe`, which presents the user with a `bash` shell running Linux. Internally, the WSL creates a [Linux instance](#) consisting of the `init` process, which in turn creates the `bash` shell running the native Linux application `/bin/bash`. Each of these processes runs in a Windows [Pico](#) process. This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls. When the Linux application makes a system call that has no Windows equivalent, the LXSS service must provide the equivalent functionality. When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel. In some situations, Linux and Windows have system calls that are similar but not identical. When this occurs, LXSS will provide some of the functionality and will invoke the similar Windows system call to provide the remainder of the functionality. The Linux `fork()` provides an illustration of this: The Windows `CreateProcess()` system call is similar to `fork()` but does not provide exactly the same functionality. When `fork()` is invoked in WSL, the LXSS service does some of the initial work of `fork()` and then calls `CreateProcess()` to do the remainder of the work. The figure below illustrates the basic behavior of WSL.



2.9 Building and Booting an Operating System

It is possible to design, code, and implement an operating system specifically for one specific machine configuration. More commonly, however, operating systems are designed to run on any of a class of machines with a variety of peripheral configurations.

2.9.1 Operating-System Generation

Most commonly, a computer system, when purchased, has an operating system already installed. For example, you may purchase a new laptop with Windows or macOS preinstalled. But suppose you wish to replace the preinstalled operating system or add additional operating systems. Or suppose you purchase a computer without an operating system. In these latter situations, you have a few options for placing the appropriate operating system on the computer and configuring it for use.

If you are generating (or building) an operating system from scratch, you must follow these steps:

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

Configuring the system involves specifying which features will be included, and this varies by operating system. Typically, parameters describing how the system is configured is stored in a configuration file of some type, and once this file is created, it can be used in several ways.

At one extreme, a system administrator can use it to modify a copy of the operating-system source code. Then the operating system is completely compiled (known as a **system build**). Data declarations, initializations, and constants, along with compilation, produce an output-object version of the operating system that is tailored to the system described in the configuration file.

At a slightly less tailored level, the system description can lead to the selection of precompiled object modules from an existing library. These modules are linked together to form the generated operating system. This process allows the library to contain the device drivers for all supported I/O devices, but only those needed are selected and linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general and may not support different hardware configurations.

At the other extreme, it is possible to construct a system that is completely modular. Here, selection occurs at execution time rather than at compile or link time. System generation involves simply setting the parameters that describe the system configuration.

The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. For embedded systems, it is not uncommon to adopt the first approach and create an operating system for a specific, static hardware configuration. However, most modern operating systems that support desktop and laptop computers as well as mobile devices have adopted the second approach. That is, the operating system is still generated for a specific hardware configuration, but the use of techniques such as loadable kernel modules provides modular support for dynamic changes to the system.

We now illustrate how to build a Linux system from scratch, where it is typically necessary to perform the following steps:

1. Download the Linux source code from <http://www.kernel.org>.
2. Configure the kernel using the “`make menuconfig`” command. This step generates the `.config` configuration file.
3. Compile the main kernel using the “`make`” command. The `make` command compiles the kernel based on the configuration parameters identified in the `.config` file, producing the file `vmlinuz`, which is the kernel image.
4. Compile the kernel modules using the “`make modules`” command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the `.config` file.
5. Use the command “`make modules_install`” to install the kernel modules into `vmlinuz`.
6. Install the new kernel on the system by entering the “`make install`” command.

When the system reboots, it will begin running this new operating system.

Alternatively, it is possible to modify an existing system by installing a Linux virtual machine. This will allow the host operating system (such as Windows or macOS) to run Linux. (We introduced virtualization in Section 1.7 and cover the topic more fully in Chapter 18.)

There are a few options for installing Linux as a virtual machine. One alternative is to build a virtual machine from scratch. This option is similar to building a Linux system from scratch; however, the operating system does not need to be compiled. Another approach is to use a Linux virtual machine appliance, which is an operating system that has already been built and configured. This option simply requires downloading the appliance and installing it using virtualization software such as VirtualBox or VMware. For example, to build the operating system used in the virtual machine provided with this text, the authors did the following:

1. Downloaded the Ubuntu ISO image from <https://www.ubuntu.com/>
2. Instructed the virtual machine software VirtualBox to use the ISO as the bootable medium and booted the virtual machine
3. Answered the installation questions and then installed and booted the operating system as a virtual machine

2.9.2 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The process of starting a computer by loading the kernel is known as **booting** the system. On most systems, the boot process proceeds as follows:

1. A small piece of code known as the **bootstrap program** or **boot loader** locates the kernel.
2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

In this section, we briefly describe the boot process in more detail.

Some computer systems use a multistage boot process: When the computer is first powered on, a small boot loader located in nonvolatile firmware known as **BIOS** is run. This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.

Many recent computer systems have replaced the BIOS-based boot process with **UEFI** (Unified Extensible Firmware Interface). UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. Perhaps the greatest advantage is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks. In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine—for example, inspecting memory and the CPU and discovering devices. If the diagnostics pass, the program can continue with the booting steps. The bootstrap can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system and mounts the root file system. It is only at this point is the system said to be **running**.

GRUB is an open-source bootstrap program for Linux and UNIX systems. Boot parameters for the system are set in a GRUB configuration file, which is loaded at startup. GRUB is flexible and allows changes to be made at boot time, including modifying kernel parameters and even selecting among different kernels that can be booted. As an example, the following are kernel parameters from the special Linux file `/proc/cmdline`, which is used at boot time:

```
BOOT_IMAGE=/boot/vmlinuz-4.4.0-59-generic
root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92
```

`BOOT_IMAGE` is the name of the kernel image to be loaded into memory, and `root` specifies a unique identifier of the root file system.

To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as `initramfs`. This file system contains necessary drivers and kernel modules that must be installed to support the *real* root file system (which is not in main memory). Once the kernel has started and the necessary drivers are installed, the kernel switches the root file system from the temporary RAM location to the appropriate root file system location. Finally, Linux creates the `systemd` process, the initial process in the system, and then starts other services (for example, a web server and/or database). Ultimately, the system will present the user with a login prompt. In Section 11.5.2, we describe the boot process for Windows.

It is worthwhile to note that the booting mechanism is not independent from the boot loader. Therefore, there are specific versions of the GRUB boot loader for BIOS and UEFI, and the firmware must know as well which specific bootloader is to be used.

The boot process for mobile systems is slightly different from that for traditional PCs. For example, although its kernel is Linux-based, Android does not use GRUB and instead leaves it up to vendors to provide boot loaders. The most common Android boot loader is LK (for “little kernel”). Android systems use the same compressed kernel image as Linux, as well as an initial RAM file system. However, whereas Linux discards the `initramfs` once all necessary drivers have been loaded, Android maintains `initramfs` as the root file system for the device. Once the kernel has been loaded and the root file system mounted, Android starts the `init` process and creates a number of services before displaying the home screen.

Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—provide booting into **recovery mode** or **single-user mode** for diagnosing hardware issues, fixing corrupt file systems, and even reinstalling the operating system. In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance, which we consider in the following section.

2.10 Operating-System Debugging

We have mentioned debugging from time to time in this chapter. Here, we take a closer look. Broadly, **debugging** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing processing **bottlenecks**. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

2.10.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log file** to alert system administrators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the

“core” in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process at the time of failure.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel’s memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis. Obviously, such strategies would be unnecessary for debugging ordinary user-level processes.

2.10.2 Performance Monitoring and Tuning

We mentioned earlier that performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. Tools may be characterized as providing either *per-process* or *system-wide* observations. To make these observations, tools may use one of two approaches—*counters* or *tracing*. We explore each of these in the following sections.

2.10.2.1 Counters

Operating systems keep track of system activity through a series of counters, such as the number of system calls made or the number of operations performed to a network device or disk. The following are examples of Linux tools that use counters:

Per-Process

- `ps`—reports information for a single process or selection of processes
- `top`—reports real-time statistics for current processes

System-Wide

- `vmstat`—reports memory-usage statistics
- `netstat`—reports statistics for network interfaces
- `iostat`—reports I/O usage for disks

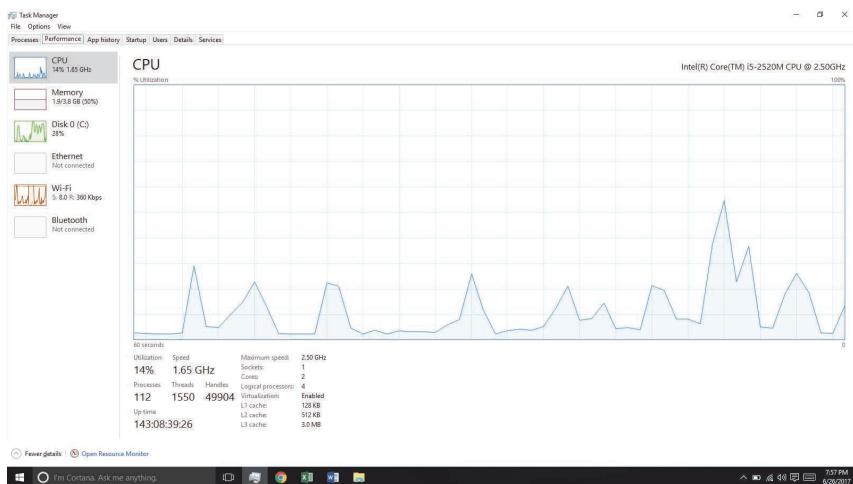


Figure 2.19 The Windows 10 task manager.

Most of the counter-based tools on Linux systems read statistics from the /proc file system. /proc is a “pseudo” file system that exists only in kernel memory and is used primarily for querying various per-process as well as kernel statistics. The /proc file system is organized as a directory hierarchy, with the process (a unique integer value assigned to each process) appearing as a subdirectory below /proc. For example, the directory entry /proc/2155 would contain per-process statistics for the process with an ID of 2155. There are /proc entries for various kernel statistics as well. In both this chapter and Chapter 3, we provide programming projects where you will create and access the /proc file system.

Windows systems provide the **Windows Task Manager**, a tool that includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager in Windows 10 appears in Figure 2.19.

2.10.3 Tracing

Whereas counter-based tools simply inquire on the current value of certain statistics that are maintained by the kernel, tracing tools collect data for a specific event—such as the steps involved in a system-call invocation.

The following are examples of Linux tools that trace events:

Per-Process

- **strace**—traces system calls invoked by a process
- **gdb**—a source-level debugger

System-Wide

- **perf**—a collection of Linux performance tools
- **tcpdump**—collects network packets

Kernighan's Law

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Making operating systems easier to understand, debug, and tune as they run is an active area of research and practice. A new generation of kernel-enabled performance analysis tools has made significant improvements in how this goal can be achieved. Next, we discuss BCC, a toolkit for dynamic kernel tracing in Linux.

2.10.4 BCC

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument their interactions. For that toolset to be truly useful, it must be able to debug any area of a system, including areas that were not written with debugging in mind, and do so without affecting system reliability. This toolset must also have a minimal performance impact—ideally it should have no impact when not in use and a proportional impact during use. The BCC toolkit meets these requirements and provides a dynamic, secure, low-impact debugging environment.

BCC (BPF Compiler Collection) is a rich toolkit that provides tracing features for Linux systems. BCC is a front-end interface to the eBPF (extended Berkeley Packet Filter) tool. The BPF technology was developed in the early 1990s for filtering traffic across a computer network. The “extended” BPF (eBPF) added various features to BPF. eBPF programs are written in a subset of C and are compiled into eBPF instructions, which can be dynamically inserted into a running Linux system. The eBPF instructions can be used to capture specific events (such as a certain system call being invoked) or to monitor system performance (such as the time required to perform disk I/O). To ensure that eBPF instructions are well behaved, they are passed through a **verifie** before being inserted into the running Linux kernel. The verifier checks to make sure that the instructions do not affect system performance or security.

Although eBPF provides a rich set of features for tracing within the Linux kernel, it traditionally has been very difficult to develop programs using its C interface. BCC was developed to make it easier to write tools using eBPF by providing a front-end interface in Python. A BCC tool is written in Python and it embeds C code that interfaces with the eBPF instrumentation, which in turn interfaces with the kernel. The BCC tool also compiles the C program into eBPF instructions and inserts it into the kernel using either probes or tracepoints, two techniques that allow tracing events in the Linux kernel.

The specifics of writing custom BCC tools are beyond the scope of this text, but the BCC package (which is installed on the Linux virtual machine we provide) provides a number of existing tools that monitor several areas

of activity in a running Linux kernel. As an example, the BCC `disksnoop` tool traces disk I/O activity. Entering the command

```
./disksnoop.py
```

generates the following example output:

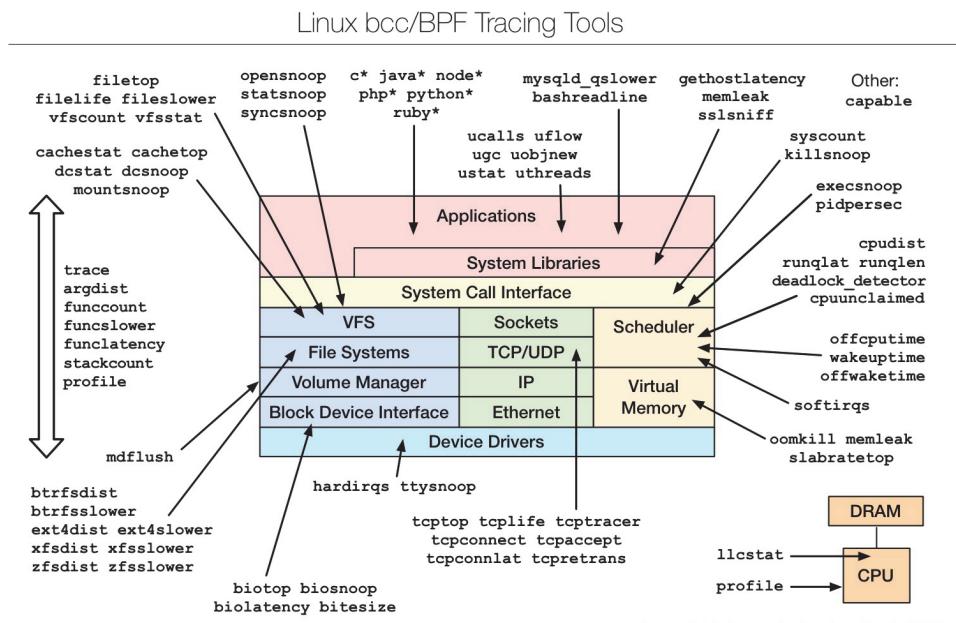
TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

This output tells us the timestamp when the I/O operation occurred, whether the I/O was a Read or Write operation, and how many bytes were involved in the I/O. The final column reflects the duration (expressed as latency or LAT) in milliseconds of the I/O.

Many of the tools provided by BCC can be used for specific applications, such as MySQL databases, as well as Java and Python programs. Probes can also be placed to monitor the activity of a specific process. For example, the command

```
./opensnoop -p 1225
```

will trace `open()` system calls performed only by the process with an identifier of 1225.



<https://github.com/iovisor/bcc#tools> 2017

Figure 2.20 The BCC and eBPF tracing tools.

What makes BCC especially powerful is that its tools can be used on live production systems that are running critical applications without causing harm to the system. This is particularly useful for system administrators who must monitor system performance to identify possible bottlenecks or security exploits. Figure 2.20 illustrates the wide range of tools currently provided by BCC and eBPF and their ability to trace essentially any area of the Linux operating system. BCC is a rapidly changing technology with new features constantly being added.

2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A monolithic operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A layered operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some suc-

cess, this approach is generally not ideal for designing operating systems due to performance problems.

- The microkernel approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.
- A modular approach for designing operating systems provides operating-system services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as hybrid systems using a combination of a monolithic kernel and modules.
- A boot loader loads an operating system into memory, performs initialization, and begins system execution.
- The performance of an operating system can be monitored using either counters or tracing. Counters are a collection of system-wide or per-process statistics, while tracing follows the execution of a program through the operating system.

Practice Exercises

- 2.1 What is the purpose of system calls?
- 2.2 What is the purpose of the command interpreter? Why is it usually separate from the kernel?
- 2.3 What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?
- 2.4 What is the purpose of system programs?
- 2.5 What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?
- 2.6 List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.
- 2.7 Why do some systems store the operating system in firmware, while others store it on disk?
- 2.8 How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

Further Reading

[Bryant and O'Hallaron (2015)] provide an overview of computer systems, including the role of the linker and loader. [Atlidakis et al. (2016)] discuss POSIX system calls and how they relate to modern operating systems. [Levin (2013)] covers the internals of both macOS and iOS, and [Levin (2015)] describes details of the Android system. Windows 10 internals are covered in [Russinovich et al. (2017)]. BSD UNIX is described in [McKusick et al. (2015)]. [Love (2010)] and

[Mauerer (2008)] thoroughly discuss the Linux kernel. Solaris is fully described in [McDougall and Mauro (2007)].

Linux source code is available at <http://www.kernel.org>. The Ubuntu ISO image is available from <https://www.ubuntu.com/>.

Comprehensive coverage of Linux kernel modules can be found at <http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf>. [Ward (2015)] and <http://www.ibm.com/developerworks/linux/library/l-linuxboot/> describe the Linux boot process using GRUB. Performance tuning—with a focus on Linux and Solaris systems—is covered in [Gregg (2014)]. Details for the BCC toolkit can be found at <https://github.com/iovisor/bcc/#tools>.

Bibliography

- [Atlidakis et al. (2016)] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, “POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing” (2016), pages 19:1–19:17.
- [Bryant and O’Hallaron (2015)] R. Bryant and D. O’Hallaron, *Computer Systems: A Programmer’s Perspective*, Third Edition (2015).
- [Gregg (2014)] B. Gregg, *Systems Performance—Enterprise and the Cloud*, Pearson (2014).
- [Levin (2013)] J. Levin, *Mac OS X and iOS Internals to the Apple’s Core*, Wiley (2013).
- [Levin (2015)] J. Levin, *Android Internals—A Confectioner’s Cookbook. Volume I* (2015).
- [Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).
- [Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).
- [McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).
- [McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System—Second Edition*, Pearson (2015).
- [Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).
- [Ward (2015)] B. Ward, *How LINUX Works—What Every Superuser Should Know*, Second Edition, No Starch Press (2015).

Chapter 2 Exercises

- 2.9 The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories, and discuss how they differ.
- 2.10 Describe three general methods for passing parameters to the operating system.
- 2.11 Describe how you could obtain a statistical profile of the amount of time a program spends executing different sections of its code. Discuss the importance of obtaining such a statistical profile.
- 2.12 What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?
- 2.13 Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?
- 2.14 Describe why Android uses ahead-of-time (AOT) rather than just-in-time (JIT) compilation.
- 2.15 What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?
- 2.16 Contrast and compare an application programming interface (API) and an application binary interface (ABI).
- 2.17 Why is the separation of mechanism and policy desirable?
- 2.18 It is sometimes difficult to achieve a layered approach if two components of the operating system are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.
- 2.19 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?
- 2.20 What are the advantages of using loadable kernel modules?
- 2.21 How are iOS and Android similar? How are they different?
- 2.22 Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.
- 2.23 The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and system-performance optimization.

Programming Problems

- 2.24** In Section 2.3, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write this program using either the POSIX or Windows API. Be sure to include all necessary error checking, including ensuring that the source file exists.

Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls. Linux systems provide the `strace` utility, and macOS systems use the `dtruss` command. (The `dtruss` command, which actually is a front end to `dtrace`, requires admin privileges, so it must be run using `sudo`.) These tools can be used as follows (assume that the name of the executable file is `FileCopy`):

Linux:

```
strace ./FileCopy
```

macOS:

```
sudo dtruss ./FileCopy
```

Since Windows systems do not provide such a tool, you will have to trace through the Windows version of this program using a debugger.

Programming Projects

Introduction to Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. You will then modify the kernel module so that it creates an entry in the `/proc` file system. The project can be completed using the Linux virtual machine that is available with this text. Although you may use any text editor to write these C programs, you will have to use the *terminal* application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing *kernel code* that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

I. Kernel Modules Overview

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

You can list all kernel modules that are currently loaded by entering the command

```
lsmod
```

This command will list the current kernel modules in three columns: name, size, and where the module is being used.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
    printk(KERN_INFO "Loading Kernel Module\n");

    return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
    printk(KERN_INFO "Removing Kernel Module\n");
}

/* Macros for registering module entry and exit points. */
MODULE_INIT(simple_init);
MODULE_EXIT(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

Figure 2.21 Kernel module simple.c.

The program in Figure 2.21 (named `simple.c` and available with the source code for this text) illustrates a very basic kernel module that prints appropriate messages when it is loaded and unloaded.

The function `simple_init()` is the **module entry point**, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the `simple_exit()` function is the **module exit point**—the function that is called when the module is removed from the kernel.

The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns `void`. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module_init(simple_init)

module_exit(simple_exit)
```

Notice in the figure how the module entry and exit point functions make calls to the `printk()` function. `printk()` is the kernel equivalent of `printf()`, but its output is sent to a kernel log buffer whose contents can be read by the `dmesg` command. One difference between `printf()` and `printk()` is that `printk()` allows us to specify a priority flag, whose values are given in the `<linux/printk.h>` include file. In this instance, the priority is `KERN_INFO`, which is defined as an *informational* message.

The final lines—`MODULE_LICENSE()`, `MODULE_DESCRIPTION()`, and `MODULE_AUTHOR()`—represent details regarding the software license, description of the module, and author. For our purposes, we do not require this information, but we include it because it is standard practice in developing kernel modules.

This kernel module `simple.c` is compiled using the `Makefile` accompanying the source code with this project. To compile the module, enter the following on the command line:

```
make
```

The compilation produces several files. The file `simple.ko` represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

II. Loading and Removing Kernel Modules

Kernel modules are loaded using the `insmod` command, which is run as follows:

```
sudo insmod simple.ko
```

To check whether the module has loaded, enter the `lsmod` command and search for the module `simple`. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message "Loading Module."

Removing the kernel module involves invoking the `rmmmod` command (notice that the `.ko` suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the `dmesg` command to ensure the module has been removed.

Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

```
sudo dmesg -c
```

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using `dmesg` to ensure that you have followed the steps properly.

As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications. For example, the Linux include file `<linux/hash.h>` defines several hashing functions for use within the kernel. This file also defines the constant value `GOLDEN_RATIO_PRIME` (which is defined as an `unsigned long`). This value can be printed out as follows:

```
 printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
```

As another example, the include file `<linux/gcd.h>` defines the following function

```
unsigned long gcd(unsigned long a, unsigned b);
```

which returns the greatest common divisor of the parameters `a` and `b`.

Once you are able to correctly load and unload your module, complete the following additional steps:

- 1.** Print out the value of `GOLDEN_RATIO_PRIME` in the `simple_init()` function.
- 2.** Print out the greatest common divisor of 3,300 and 24 in the `simple_exit()` function.

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running `make` regularly. Be sure to load and remove the kernel module and check the kernel log buffer using `dmesg` to ensure that your changes to `simple.c` are working properly.

In Section 1.4.3, we described the role of the timer as well as the timer interrupt handler. In Linux, the rate at which the timer ticks (the **tick rate**) is the value `HZ` defined in `<asm/param.h>`. The value of `HZ` determines the frequency of the timer interrupt, and its value varies by machine type and architecture. For example, if the value of `HZ` is 100, a timer interrupt occurs 100 times per second, or every 10 milliseconds. Additionally, the kernel keeps track of the global variable `jiffies`, which maintains the number of timer interrupts that have occurred since the system was booted. The `jiffies` variable is declared in the file `<linux/jiffies.h>`.

- 1.** Print out the values of `jiffies` and `HZ` in the `simple_init()` function.
- 2.** Print out the value of `jiffies` in the `simple_exit()` function.

Before proceeding to the next set of exercises, consider how you can use the different values of jiffies in `simple_init()` and `simple_exit()` to determine the number of seconds that have elapsed since the time the kernel module was loaded and then removed.

III. The /proc File System

The /proc file system is a “pseudo” file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

ssize_t proc_read(struct file *file, char __user *usr_buf,
    size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
};

/* This function is called when the module is loaded. */
int proc_init(void)
{
    /* creates the /proc/hello entry */
    proc_create(PROC_NAME, 0666, NULL, &proc_ops);

    return 0;
}

/* This function is called when the module is removed. */
void proc_exit(void)
{
    /* removes the /proc/hello entry */
    remove_proc_entry(PROC_NAME, NULL);
}
```

Figure 2.22 The /proc file-system kernel module, Part 1

This exercise involves designing kernel modules that create additional entries in the /proc file system involving both kernel statistics and information related

to specific processes. The entire program is included in Figure 2.22 and Figure 2.23.

We begin by describing how to create a new entry in the /proc file system. The following program example (named `hello.c` and available with the source code for this text) creates a /proc entry named /proc/hello. If a user enters the command

```
cat /proc/hello
```

the infamous Hello World message is returned.

```
/* This function is called each time /proc/hello is read */
ssize_t proc_read(struct file *file, char __user *usr_buf,
    size_t count, loff_t *pos)
{
    int rv = 0;
    char buffer[BUFFER_SIZE];
    static int completed = 0;

    if (completed) {
        completed = 0;
        return 0;
    }

    completed = 1;

    rv = sprintf(buffer, "Hello World\n");

    /* copies kernel space buffer to user space usr_buf */
    copy_to_user(usr_buf, buffer, rv);

    return rv;
}
module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```

Figure 2.23 The /proc file system kernel module, Part 2

In the module entry point `proc_init()`, we create the new /proc/hello entry using the `proc_create()` function. This function is passed `proc_ops`, which contains a reference to a `struct file_operations`. This struct initial-

izes the `.owner` and `.read` members. The value of `.read` is the name of the function `proc_read()` that is to be called whenever `/proc/hello` is read.

Examining this `proc_read()` function, we see that the string "Hello World\n" is written to the variable `buffer` where `buffer` exists in kernel memory. Since `/proc/hello` can be accessed from user space, we must copy the contents of `buffer` to user space using the kernel function `copy_to_user()`. This function copies the contents of kernel memory `buffer` to the variable `usr_buf`, which exists in user space.

Each time the `/proc/hello` file is read, the `proc_read()` function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string "Hello World\n") that is to go into the corresponding `/proc/hello` file.

Finally, notice that the `/proc/hello` file is removed in the module exit point `proc_exit()` using the function `remove_proc_entry()`.

IV. Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a `/proc` file named `/proc/jiffies` that reports the current value of `jiffies` when the `/proc/jiffies` file is read, such as with the command

```
cat /proc/jiffies
```

Be sure to remove `/proc/jiffies` when the module is removed.

2. Design a kernel module that creates a `proc` file named `/proc/seconds` that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of `jiffies` as well as the `HZ` rate. When a user enters the command

```
cat /proc/seconds
```

your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove `/proc/seconds` when the module is removed.

Part Two

Process Management

A process is a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Modern operating systems support processes having multiple threads of control. On systems with multiple hardware processing cores, these threads can run in parallel.

One of the most important aspects of an operating system is how it schedules threads onto available processing cores. Several choices for designing CPU schedulers are available to programmers.

Processes



Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern computing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are best done in user space, rather than within the kernel. A system therefore consists of a collection of processes, some executing user code, others executing operating system code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. In this chapter, you will read about what processes are, how they are represented in an operating system, and how they work.

CHAPTER OBJECTIVES

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.
- Describe client–server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

3.1 Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. Early computers were batch systems that executed **jobs**, followed by the emergence of time-shared systems that ran **user programs**, or **tasks**. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. And even if a computer can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them **processes**.

Although we personally prefer the more contemporary term *process*, the term *job* has historical significance, as much of operating system theory and terminology was developed during a time when the major activity of operating systems was job processing. Therefore, in some appropriate instances we use *job* when describing the role of the operating system. As an example, it would be misleading to avoid the use of commonly accepted terms that include the word *job* (such as *job scheduling*) simply because *process* has superseded *job*.

3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor's registers. The **memory layout** of a process is typically divided into multiple **sections**, and is shown in Figure 3.1. These sections include:

- **Text section**—the executable code
- **Data section**—global variables

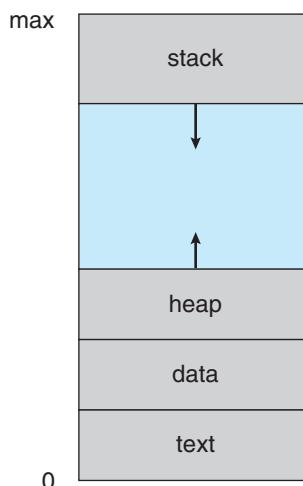


Figure 3.1 Layout of a process in memory.

- **Heap section**—memory that is dynamically allocated during program run time
- **Stack section**—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

Notice that the sizes of the text and data sections are fixed, as their sizes do not change during program run time. However, the stack and heap sections can shrink and grow dynamically during program execution. Each time a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack. Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the stack and heap sections grow *toward* one another, the operating system must ensure they do not *overlap* one another.

We emphasize that a program by itself is not a process. A program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file**). In contrast, a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in `prog.exe` or `a.out`).

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs. We discuss such matters in Section 3.4.

Note that a process can itself be an execution environment for other code. The Java programming environment provides a good example. In most circumstances, an executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code. For example, to run the compiled Java program `Program.class`, we would enter

```
java Program
```

The command `java` runs the JVM as an ordinary process, which in turns executes the Java program `Program` in the virtual machine. The concept is the same as simulation, except that the code, instead of being written for a different instruction set, is written in the Java language.

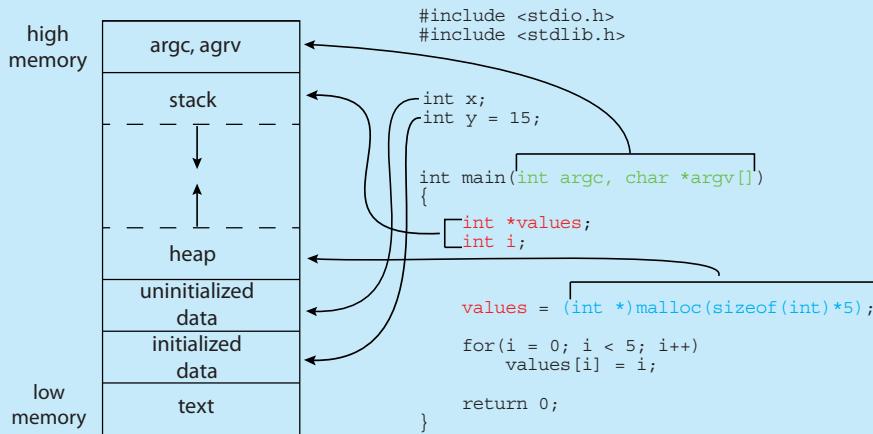
3.1.2 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to *block started by symbol*.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.

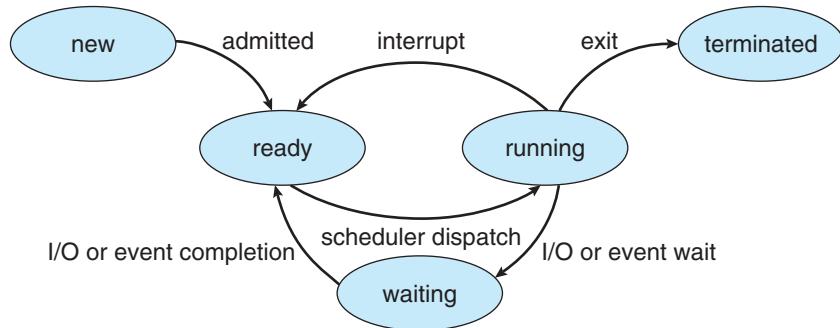


Figure 3.2 Diagram of process state.

- **Terminated.** The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor core at any instant. Many processes may be *ready* and *waiting*, however. The state diagram corresponding to these states is presented in Figure 3.2.

3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.

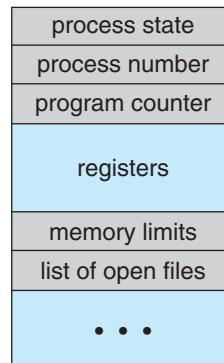


Figure 3.3 Process control block (PCB).

- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)
- **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 9).
- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

3.1.4 Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads. Chapter 4 explores threads in detail.

3.2 Process Scheduling

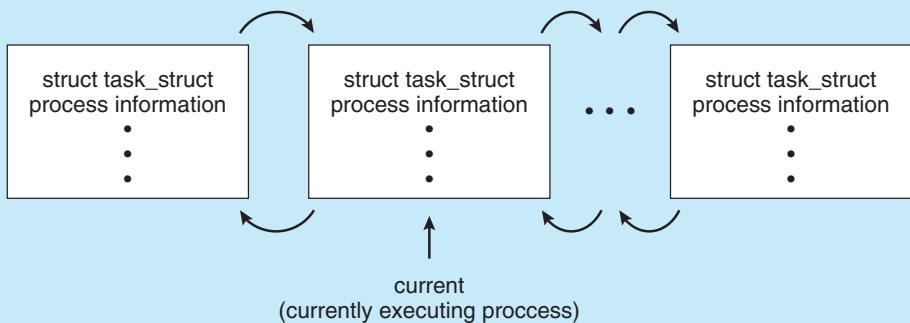
The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time.

PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`, which is found in the `<include/linux/sched.h>` include file in the kernel source-code directory. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and a list of its children and siblings. (A process's `parent` is the process that created it; its `children` are any processes that it creates. Its `siblings` are children with the same parent process.) Some of these fields include:

```
long state;           /* state of the process */
struct sched_entity se;    /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm;      /* address space */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`. The kernel maintains a pointer—`current`—to the process currently executing on the system, as shown below:



As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time. If there are more processes than cores, excess processes will have

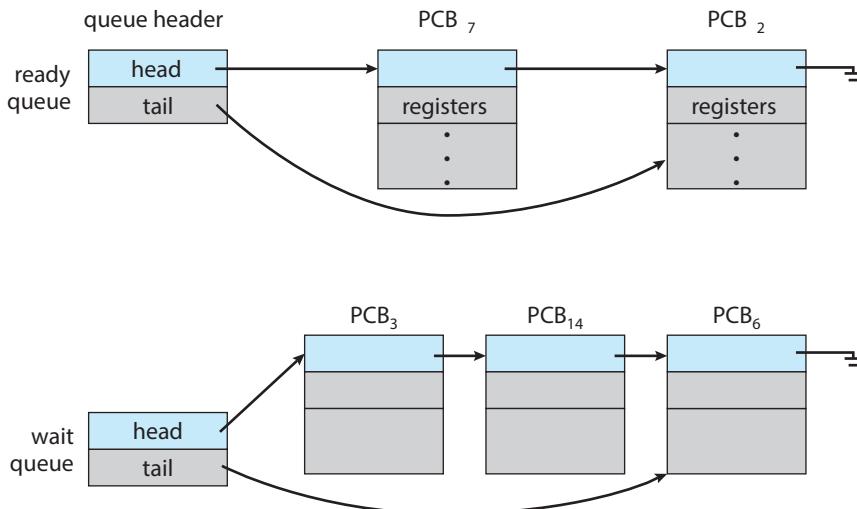


Figure 3.4 The ready queue and wait queues.

to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the **degree of multiprogramming**.

Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.

3.2.1 Scheduling Queues

As processes enter the system, they are put into a **ready queue**, where they are ready and waiting to execute on a CPU's core. This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available. Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a **wait queue** (Figure 3.4).

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 3.5. Two types of queues are present: the ready queue and a set of wait queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated a CPU core and is executing, one of several events could occur:

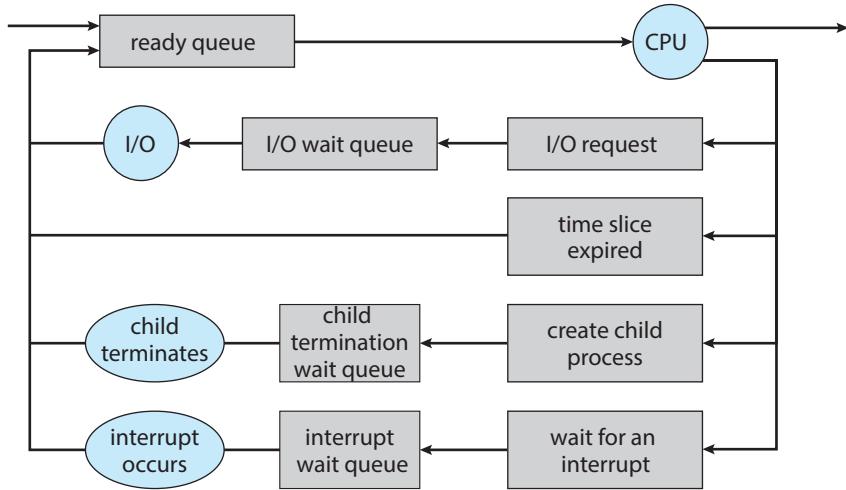


Figure 3.5 Queueing-diagram representation of process scheduling.

- The process could issue an I/O request and then be placed in an I/O wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

3.2.2 CPU Scheduling

A process migrates among the ready queue and various wait queues throughout its lifetime. The role of the **CPU scheduler** is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request. Although a CPU-bound process will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to run. Therefore, the CPU scheduler executes at least once every 100 milliseconds, although typically much more frequently.

Some operating systems have an intermediate form of scheduling, known as **swapping**, whose key idea is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is known as **swapping** because a process can be "swapped out"

from memory to disk, where its current status is saved, and later “swapped in” from disk back to memory, where its status is restored. Swapping is typically only necessary when memory has been overcommitted and must be freed up. Swapping is discussed in Chapter 9.

3.2.3 Context Switch

As mentioned in Section 1.2.1, interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current context of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch** and is illustrated in Figure 3.6. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the

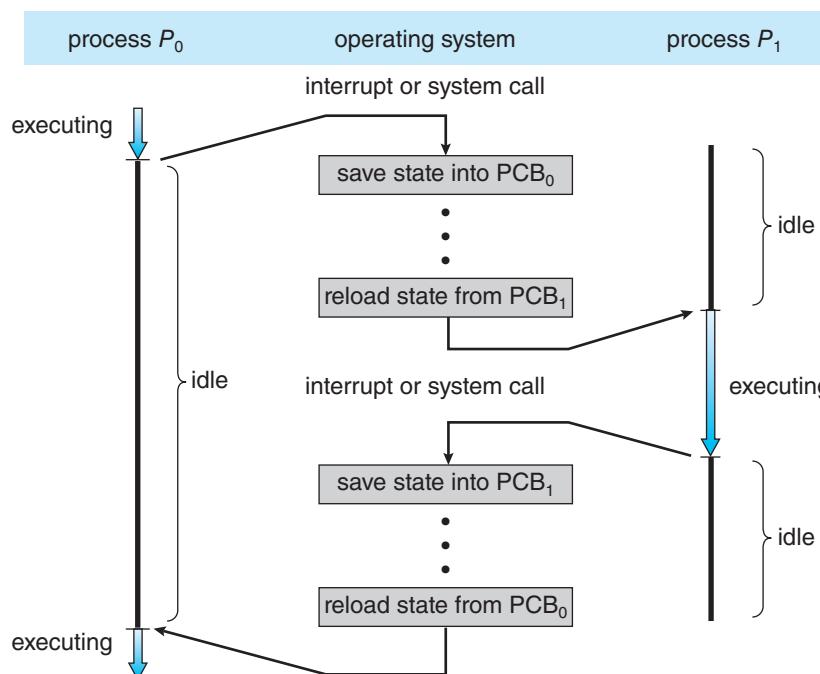


Figure 3.6 Diagram showing context switch from process to process.

MULTITASKING IN MOBILE SYSTEMS

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application ran in the foreground while all other user applications were suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple provided a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the **foreground** application is the application currently open and appearing on the display. The **background** application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provided support for multitasking, thus allowing a process to run in the background without being suspended. However, it was limited and only available for a few application types. As hardware for mobile devices began to offer larger memory capacities, multiple processing cores, and greater battery life, subsequent versions of iOS began to support richer functionality for multitasking with fewer restrictions. For example, the larger screen on iPad tablets allowed running two foreground apps at the same time, a technique known as **split-screen**.

Since its origins, Android has supported multitasking and does not place constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a **service**, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio data to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. As we will see in Chapter 9, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

3.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

3.3.1 Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

Figure 3.7 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term *process* rather loosely in this situation, as Linux prefers the term *task* instead.) The **systemd** process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots. Once the system has booted, the **systemd** process creates processes which provide additional services such as a web or print server, an ssh server, and the like. In Figure 3.7, we see two children of **systemd**—**logind** and **sshd**. The **logind** process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the **bash** shell, which has been assigned pid 8416. Using the **bash** command-line interface, this user has created the process **ps** as well as the **vim** editor. The **sshd** process is responsible for managing clients that connect to the system by using **ssh** (which is short for *secure shell*).

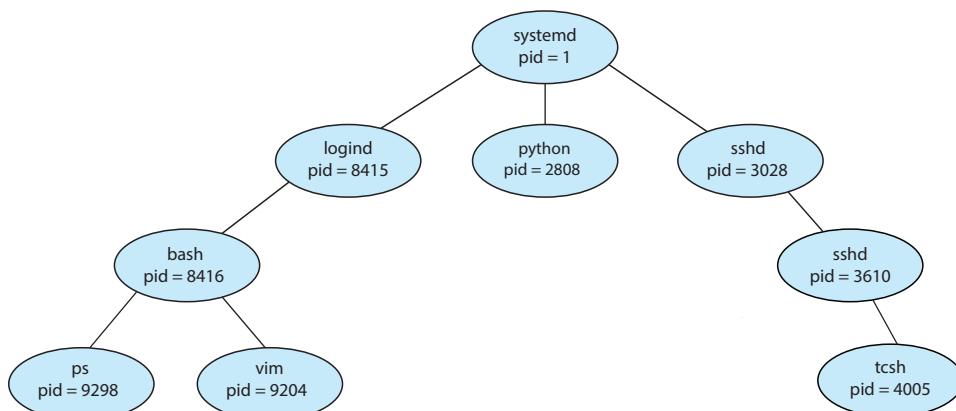


Figure 3.7 A tree of processes on a typical Linux system.

THE *init* AND *systemd* PROCESSES

Traditional UNIX systems identify the process *init* as the root of all child processes. *init* (also known as **System V init**) is assigned a pid of 1, and is the first process created when the system is booted. On a process tree similar to what is shown in Figure 3.7, *init* is at the root.

Linux systems initially adopted the System V *init* approach, but recent distributions have replaced it with *systemd*. As described in Section 3.3.1, *systemd* serves as the system's initial process, much the same as System V *init*; however it is much more flexible, and can provide more services, than *init*.

On UNIX and Linux systems, we can obtain a listing of processes by using the `ps` command. For example, the command

```
ps -el
```

will list complete information for all processes currently active in the system. A process tree similar to the one shown in Figure 3.7 can be constructed by recursively tracing parent processes all the way to the *systemd* process. (In addition, Linux systems provide the `pstree` command, which displays a tree of all processes in the system.)

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file—say, `hw1.c`—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file `hw1.c`. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, `hw1.c` and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a `fork()` system call, one of the two processes typically uses the `exec()` system call to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    execl("/bin/ls", "ls", NULL);
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
}

return 0;
}
```

Figure 3.8 Creating a separate process using the UNIX `fork()` system call.

its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child. Because the call to `exec()` overlays the process's address space with a new program, `exec()` does not return control unless an error occurs.

The C program shown in Figure 3.8 illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of the variable `pid` for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual `pid` of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execlp()` system call (`execlp()` is a version of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`), the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call. This is also illustrated in Figure 3.9.

Of course, there is nothing to prevent the child from *not* invoking `exec()` and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

As an alternative example, we next consider process creation in Windows. Processes are created in the Windows API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

The C program shown in Figure 3.10 illustrates the `CreateProcess()` function, which creates a child process that loads the application `mspaint.exe`. We opt for many of the default values of the ten parameters passed to `CreateProcess()`. Readers interested in pursuing the details of process creation and management in the Windows API are encouraged to consult the bibliographical notes at the end of this chapter.

The two parameters passed to the `CreateProcess()` function are instances of the `STARTUPINFO` and `PROCESS_INFORMATION` structures. `STARTUPINFO` specifies many properties of the new process, such as window

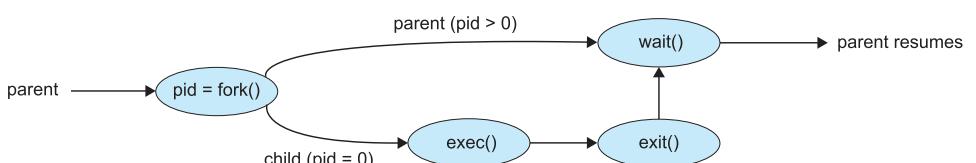


Figure 3.9 Process creation using the `fork()` system call.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

/* create child process */
if (!CreateProcess(NULL, /* use command line */
"C:\\\\WINDOWS\\\\system32\\\\mspaint.exe", /* command */
NULL, /* don't inherit process handle */
NULL, /* don't inherit thread handle */
FALSE, /* disable handle inheritance */
0, /* no creation flags */
NULL, /* use parent's environment block */
NULL, /* use parent's existing directory */
&si,
&pi))
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

Figure 3.10 Creating a separate process using the Windows API.

size and appearance and handles to standard input and output files. The `PROCESS_INFORMATION` structure contains a handle and the identifiers to the newly created process and its thread. We invoke the `ZeroMemory()` function to allocate memory for each of these structures before proceeding with `CreateProcess()`.

The first two parameters passed to `CreateProcess()` are the application name and command-line parameters. If the application name is `NULL` (as it is in this case), the command-line parameter specifies the application to load.

In this instance, we are loading the Microsoft Windows `mspaint.exe` application. Beyond these two initial parameters, we use the default parameters for inheriting process and thread handles as well as specifying that there will be no creation flags. We also use the parent's existing environment block and starting directory. Last, we provide two pointers to the `STARTUPINFO` and `PROCESS_INFORMATION` structures created at the beginning of the program. In Figure 3.8, the parent process waits for the child to complete by invoking the `wait()` system call. The equivalent of this in Windows is `WaitForSingleObject()`, which is passed a handle of the child process—`pi.hProcess`—and waits for this process to complete. Once the child process exits, control returns from the `WaitForSingleObject()` function in the parent process.

3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the `wait()` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user—or a misbehaving application—could arbitrarily kill another user's processes. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the `exit()` system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

In fact, under normal termination, `exit()` will be called either directly (as shown above) or indirectly, as the C run-time library (which is added to UNIX executable files) will include a call to `exit()` by default.

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

Now consider what would happen if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. Traditional UNIX systems addressed this scenario by assigning the `init` process as the new parent to orphan processes. (Recall from Section 3.3.1 that `init` serves as the root of the process hierarchy in UNIX systems.) The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Although most Linux systems have replaced `init` with `systemd`, the latter process can still serve the same role, although Linux also allows processes other than `systemd` to inherit orphan processes and manage their termination.

3.3.2.1 Android Process Hierarchy

Because of resource constraints such as limited memory, mobile operating systems may have to terminate existing processes to reclaim limited system resources. Rather than terminating an arbitrary process, Android has identified an **importance hierarchy** of processes, and when the system must terminate a process to make resources available for a new, or more important, process, it terminates processes in order of increasing importance. From most to least important, the hierarchy of process classifications is as follows:

- **Foreground process**—The current process visible on the screen, representing the application the user is currently interacting with
- **Visible process**—A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process)

- **Service process**—A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music)
- **Background process**—A process that may be performing an activity but is not apparent to the user.
- **Empty process**—A process that holds no active components associated with any application

If system resources must be reclaimed, Android will first terminate empty processes, followed by background processes, and so forth. Processes are assigned an importance ranking, and Android attempts to assign a process as high a ranking as possible. For example, if a process is providing a service and is also visible, it will be assigned the more-important visible classification.

Furthermore, Android development practices suggest following the guidelines of the process life cycle. When these guidelines are followed, the state of a process will be saved prior to termination and resumed at its saved state if the user navigates back to the application.

3.4 Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is *independent* if it does not share data with any other processes executing in the system. A process is *cooperating* if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing.** Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 2.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data—that is, send data to and receive data from each other. There are two fundamental models of interprocess communication: **shared memory** and **message passing**. In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model,

MULTIPROCESS ARCHITECTURE—CHROME BROWSER

Many websites contain active content, such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites, a user need only click on the appropriate tab. This arrangement is illustrated below:



A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites—crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

- The **browser** process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.
- **Renderer** processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.
- A **plug-in** process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

The advantage of the multiprocess approach is that websites run in isolation from one another. If one website crashes, only its renderer process is affected; all other processes remain unharmed. Furthermore, renderer processes run in a **sandbox**, which means that access to disk and network I/O is restricted, minimizing the effects of any security exploits.

communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.11.

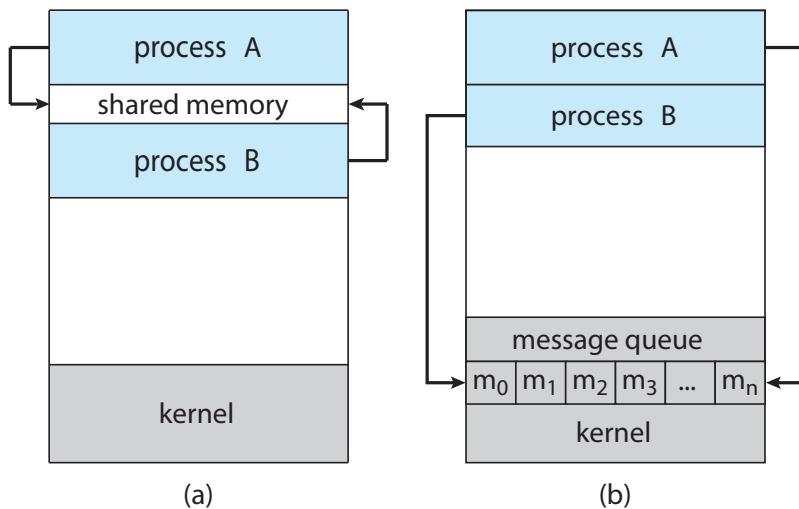


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory. (Although there are systems that provide distributed shared memory, we do not consider them in this text.) Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

In Section 3.5 and Section 3.6 we explore shared-memory and message-passing systems in more detail.

3.5 IPC in Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader. The producer–consumer problem also provides a useful metaphor for the client–server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) web content such as HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Let's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in == out`; the buffer is full when $((in + 1) \% \text{BUFFER_SIZE}) == out$.

The code for the producer process is shown in Figure 3.12, and the code for the consumer process is shown in Figure 3.13. The producer process has a local variable `next_produced` in which the new item to be produced is stored. The consumer process has a local variable `next_consumed` in which the item to be consumed is stored.

This scheme allows at most `BUFFER_SIZE - 1` items in the buffer at the same time. We leave it as an exercise for you to provide a solution in which `BUFFER_SIZE` items can be in the buffer at the same time. In Section 3.7.1, we illustrate the POSIX API for shared memory.

```

item next_produced;

while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

```

Figure 3.12 The producer process using shared memory.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently. In Chapter 6 and Chapter 7, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

3.6 IPC in Message-Passing Systems

In Section 3.5, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

```

item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_consumed */
}

```

Figure 3.13 The consumer process using shared memory.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

send(message)

and

receive(message)

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating-system design.

If processes P and Q want to communicate, they must send messages to and receive messages from each other: a *communication link* must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 19) but rather with its logical implementation. Here are several methods for logically implementing a link and the `send()`/`receive()` operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

We look at issues related to each of these features next.

3.6.1 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send()` and `receive()` primitives are defined as:

- `send(P, message)`—Send a message to process P .
- `receive(Q, message)`—Receive a message from process Q .

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits *symmetry* in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs *asymmetry* in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)`—Send a message to process P.
- `receive(id, message)`—Receive a message from any process. The variable `id` is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such *hard-coding* techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With *indirect communication*, the messages are sent to and received from *mailboxes*, or *ports*. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The `send()` and `receive()` primitives are defined as follows:

- `send(A, message)`—Send a message to mailbox A.
- `receive(A, message)`—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Now suppose that processes P_1 , P_2 , and P_3 all share mailbox A. Process P_1 sends a message to A, while both P_2 and P_3 execute a `receive()` from A. Which process will receive the message sent by P_1 ? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.

- Allow at most one process at a time to execute a `receive()` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3 , but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, *round robin*, where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

3.6.2 Synchronization

Communication between processes takes place through calls to `send()` and `receive()` primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

- **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send.** The sending process sends the message and resumes operation.
- **Blocking receive.** The receiver blocks until a message is available.
- **Nonblocking receive.** The receiver retrieves either a valid message or a null.

```

message next_produced;

while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}

```

Figure 3.14 The producer process using message passing.

Different combinations of `send()` and `receive()` are possible. When both `send()` and `receive()` are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer-consumer problem becomes trivial when we use blocking `send()` and `receive()` statements. The producer merely invokes the blocking `send()` call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes `receive()`, it blocks until a message is available. This is illustrated in Figures 3.14 and 3.15.

3.6.3 Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity.** The queue has finite length n ; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without

```

message next_consumed;

while (true) {
    receive(next_consumed);

    /* consume the item in next_consumed */
}

```

Figure 3.15 The consumer process using message passing.

waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

- **Unbounded capacity.** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

3.7 Examples of IPC Systems

In this section, we explore four different IPC systems. We first cover the POSIX API for shared memory and then discuss message passing in the Mach operating system. Next, we present Windows IPC, which interestingly uses shared memory as a mechanism for providing certain types of message passing. We conclude with pipes, one of the earliest IPC mechanisms on UNIX systems.

3.7.1 POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. Here, we explore the POSIX API for shared memory.

POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the `shm_open()` system call, as follows:

```
fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

The first parameter specifies the name of the shared-memory object. Processes that wish to access this shared memory must refer to the object by this name. The subsequent parameters specify that the shared-memory object is to be created if it does not yet exist (`O_CREAT`) and that the object is open for reading and writing (`O_RDWR`). The last parameter establishes the file-access permissions of the shared-memory object. A successful call to `shm_open()` returns an integer file descriptor for the shared-memory object.

Once the object is established, the `ftruncate()` function is used to configure the size of the object in bytes. The call

```
ftruncate(fd, 4096);
```

sets the size of the object to 4,096 bytes.

Finally, the `mmap()` function establishes a memory-mapped file containing the shared-memory object. It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

The programs shown in Figure 3.16 and Figure 3.17 use the producer-consumer model in implementing shared memory. The producer establishes a shared-memory object and writes to shared memory, and the consumer reads from shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory obect */
    char *ptr;

    /* create the shared memory object */
    fd = shm_open(name,O_CREAT | O_RDWR,0666);

    /* configure the size of the shared memory object */
    ftruncate(fd, SIZE);

    /* memory map the shared memory object */
    ptr = (char *)
        mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Figure 3.16 Producer process illustrating POSIX shared-memory API.

The producer, shown in Figure 3.16, creates a shared-memory object named OS and writes the infamous string "Hello World!" to shared memory. The program memory-maps a shared-memory object of the specified size and allows writing to the object. The flag MAP_SHARED specifies that changes to the shared-memory object will be visible to all processes sharing the object. Notice that we write to the shared-memory object by calling the `sprintf()` function and writing the formatted string to the pointer `ptr`. After each write, we must increment the pointer by the number of bytes written.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory obect */
    char *ptr;

    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = (char *)
        mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Figure 3.17 Consumer process illustrating POSIX shared-memory API.

The consumer process, shown in Figure 3.17, reads and outputs the contents of the shared memory. The consumer also invokes the `shm_unlink()` function, which removes the shared-memory segment after the consumer has accessed it. We provide further exercises using the POSIX shared-memory API in the programming exercises at the end of this chapter. Additionally, we provide more detailed coverage of memory mapping in Section 13.5.

3.7.2 Mach Message Passing

As an example of message passing, we next consider the Mach operating system. Mach was especially designed for distributed systems, but was shown to be suitable for desktop and mobile systems as well, as evidenced by its inclusion in the macOS and iOS operating systems, as discussed in Chapter 2.

The Mach kernel supports the creation and destruction of multiple *tasks*, which are similar to processes but have multiple threads of control and fewer associated resources. Most communication in Mach—including all inter-task communication—is carried out by *messages*. Messages are sent to, and received from, mailboxes, which are called *ports* in Mach. Ports are finite in size and unidirectional; for two-way communication, a message is sent to one port, and a response is sent to a separate *reply* port. Each port may have multiple senders, but only one receiver. Mach uses ports to represent resources such as tasks, threads, memory, and processors, while message passing provides an object-oriented approach for interacting with these system resources and services. Message passing may occur between any two ports on the same host or on separate hosts on a distributed system.

Associated with each port is a collection of *port rights* that identify the capabilities necessary for a task to interact with the port. For example, for a task to receive a message from a port, it must have the capability `MACH_PORT_RIGHT_RECEIVE` for that port. The task that creates a port is that port’s owner, and the owner is the only task that is allowed to receive messages from that port. A port’s owner may also manipulate the capabilities for a port. This is most commonly done in establishing a reply port. For example, assume that task T_1 owns port P_1 , and it sends a message to port P_2 , which is owned by task T_2 . If T_1 expects to receive a reply from T_2 , it must grant T_2 the right `MACH_PORT_RIGHT_SEND` for port P_1 . Ownership of port rights is at the task level, which means that all threads belonging to the same task share the same port rights. Thus, two threads belonging to the same task can easily communicate by exchanging messages through the per-thread port associated with each thread.

When a task is created, two special ports—the *Task Self* port and the *Notify* port—are also created. The kernel has receive rights to the Task Self port, which allows a task to send messages to the kernel. The kernel can send notification of event occurrences to a task’s Notify port (to which, of course, the task has receive rights).

The `mach_port_allocate()` function call creates a new port and allocates space for its queue of messages. It also identifies the rights for the port. Each port right represents a *name* for that port, and a port can only be accessed via

a right. Port names are simple integer values and behave much like UNIX file descriptors. The following example illustrates creating a port using this API:

```
mach_port_t port; // the name of the port right

mach_port_allocate(
    mach_task_self(), // a task referring to itself
    MACH_PORT_RIGHT_RECEIVE, // the right for this port
    &port); // the name of the port right
```

Each task also has access to a **bootstrap port**, which allows a task to register a port it has created with a system-wide **bootstrap server**. Once a port has been registered with the bootstrap server, other tasks can look up the port in this registry and obtain rights for sending messages to the port.

The queue associated with each port is finite in size and is initially empty. As messages are sent to the port, the messages are copied into the queue. All messages are delivered reliably and have the same priority. Mach guarantees that multiple messages from the same sender are queued in first-in, first-out (FIFO) order but does not guarantee an absolute ordering. For instance, messages from two senders may be queued in any order.

Mach messages contain the following two fields:

- A fixed-size message header containing metadata about the message, including the size of the message as well as source and destination ports. Commonly, the sending thread expects a reply, so the port name of the source is passed on to the receiving task, which can use it as a “return address” in sending a reply.
- A variable-sized body containing data.

Messages may be either *simple* or *complex*. A simple message contains ordinary, unstructured user data that are not interpreted by the kernel. A complex message may contain pointers to memory locations containing data (known as “out-of-line” data) or may also be used for transferring port rights to another task. Out-of-line data pointers are especially useful when a message must pass large chunks of data. A simple message would require copying and packaging the data in the message; out-of-line data transmission requires only a pointer that refers to the memory location where the data are stored.

The function `mach_msg()` is the standard API for both sending and receiving messages. The value of one of the function’s parameters—either `MACH_SEND_MSG` or `MACH_RCV_MSG`—indicates if it is a send or receive operation. We now illustrate how it is used when a client task sends a simple message to a server task. Assume there are two ports—client and server—associated with the client and server tasks, respectively. The code in Figure 3.18 shows the client task constructing a header and sending a message to the server, as well as the server task receiving the message sent from the client.

The `mach_msg()` function call is invoked by user programs for performing message passing. `mach_msg()` then invokes the function `mach_msg_trap()`, which is a system call to the Mach kernel. Within the kernel, `mach_msg_trap()` next calls the function `mach_msg_overwrite_trap()`, which then handles the actual passing of the message.

```

#include<mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;

/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
         MACH_SEND_MSG, // sending a message
         sizeof(message), // size of message sent
         0, // maximum size of received message - unnecessary
         MACH_PORT_NULL, // name of receive port - unnecessary
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);

/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
         MACH_RCV_MSG, // sending a message
         0, // size of message sent
         sizeof(message), // maximum size of received message
         server, // name of receive port
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);

```

Figure 3.18 Example program illustrating message passing in Mach.

The send and receive operations themselves are flexible. For instance, when a message is sent to a port, its queue may be full. If the queue is not full, the message is copied to the queue, and the sending task continues. If the

port's queue is full, the sender has several options (specified via parameters to `mach_msg()`):

1. Wait indefinitely until there is room in the queue.
2. Wait at most n milliseconds.
3. Do not wait at all but rather return immediately.
4. Temporarily cache a message. Here, a message is given to the operating system to keep, even though the queue to which that message is being sent is full. When the message can be put in the queue, a notification message is sent back to the sender. Only one message to a full queue can be pending at any time for a given sending thread.

The final option is meant for server tasks. After finishing a request, a server task may need to send a one-time reply to the task that requested the service, but it must also continue with other service requests, even if the reply port for a client is full.

The major problem with message systems has generally been poor performance caused by copying of messages from the sender's port to the receiver's port. The Mach message system attempts to avoid copy operations by using virtual-memory-management techniques (Chapter 10). Essentially, Mach maps the address space containing the sender's message into the receiver's address space. Therefore, the message itself is never actually copied, as both the sender and receiver access the same memory. This message-management technique provides a large performance boost but works only for intrasystem messages.

3.7.3 Windows

The Windows operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows provides support for multiple operating environments, or *subsystems*. Application programs communicate with these subsystems via a message-passing mechanism. Thus, application programs can be considered clients of a subsystem server.

The message-passing facility in Windows is called the **advanced local procedure call (ALPC)** facility. It is used for communication between two processes on the same machine. It is similar to the standard remote procedure call (RPC) mechanism that is widely used, but it is optimized for and specific to Windows. (Remote procedure calls are covered in detail in Section 3.8.2.) Like Mach, Windows uses a port object to establish and maintain a connection between two processes. Windows uses two types of ports: **connection ports** and **communication ports**.

Server processes publish connection-port objects that are visible to all processes. When a client wants services from a subsystem, it opens a handle to the server's connection-port object and sends a connection request to that port. The server then creates a channel and returns a handle to the client. The channel consists of a pair of private communication ports: one for client–server messages, the other for server–client messages. Additionally, communication channels support a callback mechanism that allows the client and server to accept requests when they would normally be expecting a reply.

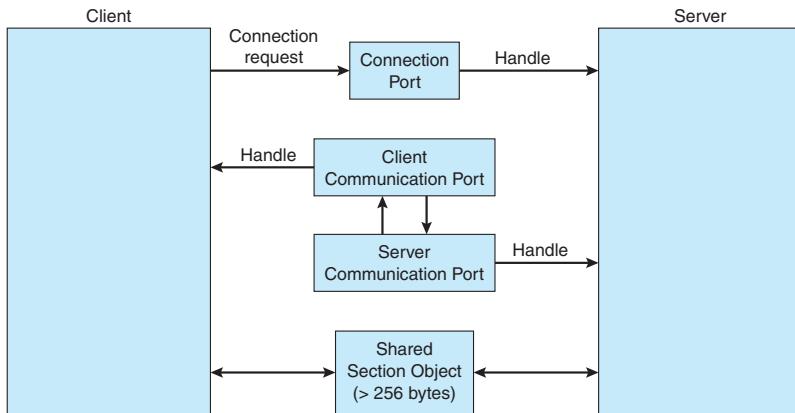


Figure 3.19 Advanced local procedure calls in Windows.

When an ALPC channel is created, one of three message-passing techniques is chosen:

1. For small messages (up to 256 bytes), the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. Larger messages must be passed through a **section object**, which is a region of shared memory associated with the channel.
3. When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

The client has to decide when it sets up the channel whether it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object. This method is more complicated than the first method listed above, but it avoids data copying. The structure of advanced local procedure calls in Windows is shown in Figure 3.19.

It is important to note that the ALPC facility in Windows is not part of the Windows API and hence is not visible to the application programmer. Rather, applications using the Windows API invoke standard remote procedure calls. When the RPC is being invoked on a process on the same system, the RPC is handled indirectly through an ALPC procedure call. Additionally, many kernel services use ALPC to communicate with client processes.

3.7.4 Pipes

A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication unidirectional?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
3. Must a relationship (such as *parent-child*) exist between the communicating processes?
4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

In the following sections, we explore two common types of pipes used on both UNIX and Windows systems: ordinary pipes and named pipes.

3.7.4.1 Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer-consumer fashion: the producer writes to one end of the pipe (the **write end**) and the consumer reads from the other end (the **read end**). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. We next illustrate constructing ordinary pipes on both UNIX and Windows systems. In both program examples, one process writes the message *Greetings* to the pipe, while the other process reads this message from the pipe.

On UNIX systems, ordinary pipes are constructed using the function

```
pipe(int fd[])
```

This function creates a pipe that is accessed through the `int fd []` file descriptors: `fd [0]` is the read end of the pipe, and `fd [1]` is the write end. UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary `read()` and `write()` system calls.

An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via `fork()`. Recall from Section 3.3.1 that a child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process. Figure 3.20 illustrates

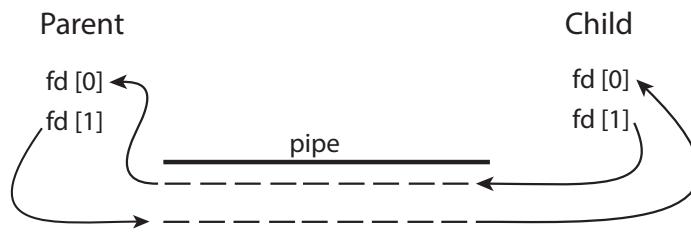


Figure 3.20 File descriptors for an ordinary pipe.

```

#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

/* Program continues in Figure 3.22 */

```

Figure 3.21 Ordinary pipe in UNIX.

the relationship of the file descriptors in the `fd` array to the parent and child processes. As this illustrates, any writes by the parent to its write end of the pipe—`fd[1]`—can be read by the child from its read end—`fd[0]`—of the pipe.

In the UNIX program shown in Figure 3.21, the parent process creates a pipe and then sends a `fork()` call creating the child process. What occurs after the `fork()` call depends on how the data are to flow through the pipe. In this instance, the parent writes to the pipe, and the child reads from it. It is important to notice that both the parent process and the child process initially close their unused ends of the pipe. Although the program shown in Figure 3.21 does not require this action, it is an important step to ensure that a process reading from the pipe can detect end-of-file (`read()` returns 0) when the writer has closed its end of the pipe.

Ordinary pipes on Windows systems are termed **anonymous pipes**, and they behave similarly to their UNIX counterparts: they are unidirectional and employ parent–child relationships between the communicating processes. In addition, reading and writing to the pipe can be accomplished with the ordinary `ReadFile()` and `WriteFile()` functions. The Windows API for creating pipes is the `CreatePipe()` function, which is passed four parameters. The parameters provide separate handles for (1) reading and (2) writing to the pipe, as well as (3) an instance of the `STARTUPINFO` structure, which is used to specify that the child process is to inherit the handles of the pipe. Furthermore, (4) the size of the pipe (in bytes) may be specified.

Figure 3.23 illustrates a parent process creating an anonymous pipe for communicating with its child. Unlike UNIX systems, in which a child process automatically inherits a pipe created by its parent, Windows requires the programmer to specify which attributes the child process will inherit. This is

```
/* create the pipe */
if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
}

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}

if (pid > 0) { /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* close the write end of the pipe */
    close(fd[WRITE_END]);
}
else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);

    /* close the read end of the pipe */
    close(fd[READ_END]);
}

return 0;
}
```

Figure 3.22 Figure 3.21, continued.

accomplished by first initializing the SECURITY_ATTRIBUTES structure to allow handles to be inherited and then redirecting the child process's handles for standard input or standard output to the read or write handle of the pipe. Since the child will be reading from the pipe, the parent must redirect the child's standard input to the read handle of the pipe. Furthermore, as the pipes are half duplex, it is necessary to prohibit the child from inheriting the write end of the

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
    HANDLE ReadHandle, WriteHandle;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    char message[BUFFER_SIZE] = "Greetings";
    DWORD written;

    /* Program continues in Figure 3.24 */
}
```

Figure 3.23 Windows anonymous pipe — parent process.

pipe. The program to create the child process is similar to the program in Figure 3.10, except that the fifth parameter is set to TRUE, indicating that the child process is to inherit designated handles from its parent. Before writing to the pipe, the parent first closes its unused read end of the pipe. The child process that reads from the pipe is shown in Figure 3.25. Before reading from the pipe, this program obtains the read handle to the pipe by invoking `GetStdHandle()`.

Note that ordinary pipes require a parent–child relationship between the communicating processes on both UNIX and Windows systems. This means that these pipes can be used only for communication between processes on the same machine.

3.7.4.2 Named Pipes

Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation differ greatly. Next, we explore named pipes in each of these systems.

Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is created with the `mkfifo()` system call and manipulated with the ordinary `open()`, `read()`, `write()`, and `close()` system calls. It will continue to exist until it is explicitly deleted

```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
    fprintf(stderr, "Create Pipe Failed");
    return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* inherit handles */
    0, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
    fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figure 3.24 Figure 3.23, continued.

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
    HANDLE Readhandle;
    CHAR buffer[BUFFER_SIZE];
    DWORD read;

    /* get the read handle of the pipe */
    ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

    /* the child reads from the pipe */
    if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
        printf("child read %s",buffer);
    else
        fprintf(stderr, "Error reading from pipe");

    return 0;
}
```

Figure 3.25 Windows anonymous pipes – child process.

from the file system. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine. If intermachine communication is required, sockets (Section 3.8.1) must be used.

Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts. Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines. Additionally, only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data. Named pipes are created with the `CreateNamedPipe()` function, and a client can connect to a named pipe using `ConnectNamedPipe()`. Communication over the named pipe can be accomplished using the `ReadFile()` and `WriteFile()` functions.

3.8 Communication in Client–Server Systems

In Section 3.4, we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client–server systems (Section 1.10.3) as well. In this section, we explore two other strategies for communication in client–server systems: sockets and

PIPES IN PRACTICE

Pipes are used quite often in the UNIX command-line environment for situations in which the output of one command serves as input to another. For example, the UNIX `ls` command produces a directory listing. For especially long directory listings, the output may scroll through several screens. The command `less` manages output by displaying only one screen of output at a time where the user may use certain keys to move forward or backward in the file. Setting up a pipe between the `ls` and `less` commands (which are running as individual processes) allows the output of `ls` to be delivered as the input to `less`, enabling the user to display a large directory listing a screen at a time. A pipe can be constructed on the command line using the `|` character. The complete command is

```
ls | less
```

In this scenario, the `ls` command serves as the producer, and its output is consumed by the `less` command.

Windows systems provide a `more` command for the DOS shell with functionality similar to that of its UNIX counterpart `less`. (UNIX systems also provide a `more` command, but in the tongue-in-cheek style common in UNIX, the `less` command in fact provides *more* functionality than `more`!) The DOS shell also uses the `|` character for establishing a pipe. The only difference is that to get a directory listing, DOS uses the `dir` command rather than `ls`, as shown below:

```
dir | more
```

remote procedure calls (RPCs). As we shall see in our coverage of RPCs, not only are they useful for client–server computing, but Android also uses remote procedures as a form of IPC between processes running on the same system.

3.8.1 Sockets

A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as SSH, FTP, and HTTP) listen to *well-known* ports (an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80). All ports below 1024 are considered *well known* and are used to implement standard services.

When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024. For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at

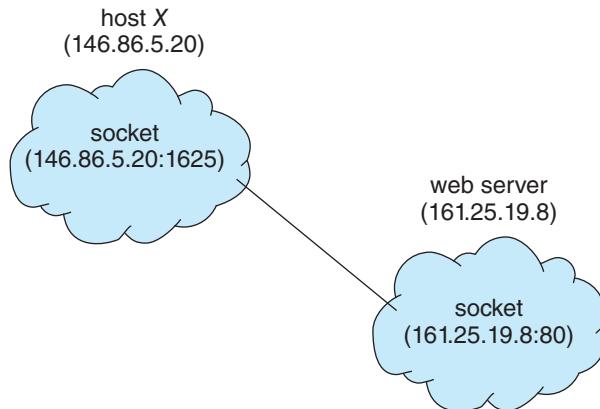


Figure 3.26 Communication using sockets.

address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.26. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the bibliographical notes at the end of the chapter.

Java provides three different types of sockets. **Connection-oriented (TCP)** sockets are implemented with the `Socket` class. **Connectionless (UDP)** sockets use the `DatagramSocket` class. Finally, the `MulticastSocket` class is a sub-class of the `DatagramSocket` class. A multicast socket allows data to be sent to multiple recipients.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary, unused number greater than 1024. When a connection is received, the server returns the date and time to the client.

The date server is shown in Figure 3.27. The server creates a `ServerSocket` that specifies that it will listen to port 6013. The server then begins listening to the port with the `accept()` method. The server blocks on the `accept()` method waiting for a client to request a connection. When a connection request is received, `accept()` returns a socket that the server can use to communicate with the client.

The details of how the server communicates with the socket are as follows. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the routine `print()` and `println()` methods for output. The

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Figure 3.27 Date server.

server process sends the date to the client, calling the method `println()`. Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. We implement such a client in the Java program shown in Figure 3.28. The client creates a `Socket` and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **loopback**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to an IP address, an actual host name, such as `www.westminstercollege.edu`, can be used as well.

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Figure 3.28 Date client.

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next subsection, we look at a higher-level method of communication: remote procedure calls (RPCs).

3.8.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, which was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 3.4, and it is usually built on top of such a system. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service.

In contrast to IPC messages, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier specifying the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A **port** in this context is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port. For instance, if a system wished to allow other systems to be able to list its current users, it would have a daemon supporting such an RPC attached to a port—say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server. The data would be received in a reply message.

The semantics of RPCs allows a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a **stub** on the client side. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and **marshals** the parameters. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique. On Windows systems, stub code is compiled from a specification written in the **Microsoft Interface Definition Language (MIDL)**, which is used for defining the interfaces between client and server programs.

Parameter marshaling addresses the issue concerning differences in data representation on the client and server machines. Consider the representation of 32-bit integers. Some systems (known as **big-endian**) store the most significant byte first, while other systems (known as **little-endian**) store the least significant byte first. Neither order is “better” per se; rather, the choice is arbitrary within a computer architecture. To resolve differences like this, many RPC systems define a machine-independent representation of data. One such representation is known as **external data representation (XDR)**. On the client side, parameter marshaling involves converting the machine-dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshaled and converted to the machine-dependent representation for the server.

Another important issue involves the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors. One way to address this problem is for the operating system to ensure that messages are acted on *exactly once*, rather than *at most once*. Most local procedure calls have the “exactly once” functionality, but it is more difficult to implement.

First, consider “at most once.” This semantic can be implemented by attaching a timestamp to each message. The server must keep a history of all the timestamps of messages it has already processed or a history large enough

to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. The client can then send a message one or more times and be assured that it only executes once.

For “exactly once,” we need to remove the risk that the server will never receive the request. To accomplish this, the server must implement the “at most once” protocol described above but must also acknowledge to the client that the RPC call was received and executed. These ACK messages are common throughout networking. The client must resend each RPC call periodically until it receives the ACK for that call.

Yet another important issue concerns the communication between a server and a client. With standard procedure calls, some form of binding takes place during link, load, or execution time (Chapter 9) so that a procedure call’s name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other, because they do not share memory.

Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach. Figure 3.29 shows a sample interaction.

The RPC scheme is useful in implementing a distributed file system (Chapter 19). Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the distributed file system port on a server on which a file operation is to take place. The message contains the disk operation to be performed. The disk operation might be `read()`, `write()`, `rename()`, `delete()`, or `status()`, corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client or be limited to a simple block request. In the latter case, several requests may be needed if a whole file is to be transferred.

3.8.2.1 Android RPC

Although RPCs are typically associated with client-server computing in a distributed system, they can also be used as a form of IPC between processes running on the same system. The Android operating system has a rich set of IPC mechanisms contained in its **binder** framework, including RPCs that allow one process to request services from another process.

Android defines an **application component** as a basic building block that provides utility to an Android application, and an app may combine multiple application components to provide functionality to an app. One such applica-

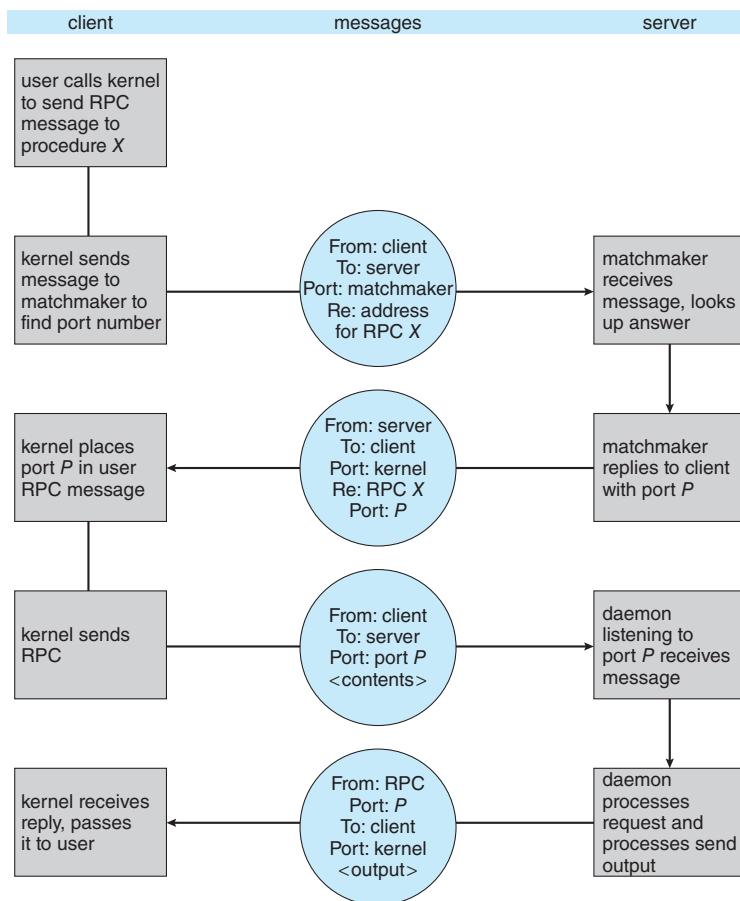


Figure 3.29 Execution of a remote procedure call (RPC).

tion component is a **service**, which has no user interface but instead runs in the background while executing long-running operations or performing work for remote processes. Examples of services include playing music in the background and retrieving data over a network connection on behalf of another process, thereby preventing the other process from blocking as the data are being downloaded. When a client app invokes the `bindService()` method of a service, that service is “bound” and available to provide client-server communication using either message passing or RPCs.

A bound service must extend the Android class `Service` and must implement the method `onBind()`, which is invoked when a client calls `bindService()`. In the case of message passing, the `onBind()` method returns a `Messenger` service, which is used for sending messages from the client to the service. The `Messenger` service is only one-way; if the service must send a reply back to the client, the client must also provide a `Messenger` service, which is contained in the `replyTo` field of the `Message` object sent to the service. The service can then send messages back to the client.

To provide RPCs, the `onBind()` method must return an interface representing the methods in the remote object that clients use to interact with the

service. This interface is written in regular Java syntax and uses the Android Interface Definition Language—AIDL—to create stub files, which serve as the client interface to remote services.

Here, we briefly outline the process required to provide a generic remote service named `remoteMethod()` using AIDL and the binder service. The interface for the remote service appears as follows:

```
/* RemoteService.aidl */
interface RemoteService
{
    boolean remoteMethod(int x, double y);
}
```

This file is written as `RemoteService.aidl`. The Android development kit will use it to generate a `.java` interface from the `.aidl` file, as well as a stub that serves as the RPC interface for this service. The server must implement the interface generated by the `.aidl` file, and the implementation of this interface will be called when the client invokes `remoteMethod()`.

When a client calls `bindService()`, the `onBind()` method is invoked on the server, and it returns the stub for the `RemoteService` object to the client. The client can then invoke the remote method as follows:

```
RemoteService service;
. . .
service.remoteMethod(3, 0.14);
```

Internally, the Android binder framework handles parameter marshaling, transferring marshaled parameters between processes, and invoking the necessary implementation of the service, as well as sending any return values back to the client process.

3.9 Summary

- A process is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
- The layout of a process in memory is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
- As a process executes, it changes state. There are four general states of a process: (1) ready, (2) running, (3) waiting, and (4) terminated.
- A process control block (PCB) is the kernel data structure that represents a process in an operating system.
- The role of the process scheduler is to select an available process to run on a CPU.
- An operating system performs a context switch when it switches from running one process to running another.

- The `fork()` and `CreateProcess()` system calls are used to create processes on UNIX and Windows systems, respectively.
- When shared memory is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
- Two processes may communicate by exchanging messages with one another using message passing. The Mach operating system uses message passing as its primary form of interprocess communication. Windows provides a form of message passing as well.
- A pipe provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent–child relationship. Named pipes are more general and allow several processes to communicate.
- UNIX systems provide ordinary pipes through the `pipe()` system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.
- Windows systems also provide two forms of pipes—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent–child relationships between the communicating processes. Named pipes offer a richer form of interprocess communication than the UNIX counterpart, FIFOs.
- Two common forms of client–server communication are sockets and remote procedure calls (RPCs). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
- The Android operating system uses RPCs as a form of interprocess communication using its binder framework.

Practice Exercises

- 3.1 Using the program shown in Figure 3.30, explain what the output will be at LINE A.
- 3.2 Including the initial parent process, how many processes are created by the program shown in Figure 3.31? 
- 3.3 Original versions of Apple’s mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.
- 3.4 Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINE A */
        return 0;
    }
}
```

Figure 3.30 What output will be at Line A?

loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

- 3.5 When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process? 
- Stack
 - Heap
 - Shared memory segments
- 3.6 Consider the “exactly once” semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether “exactly once” is still preserved.
- 3.7 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the “exactly once” semantic for execution of RPCs?

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.31 How many processes are created?

Further Reading

Process creation, management, and IPC in UNIX and Windows systems, respectively, are discussed in [Robbins and Robbins (2003)] and [Russinovich et al. (2017)]. [Love (2010)] covers support for processes in the Linux kernel, and [Hart (2005)] covers Windows systems programming in detail. Coverage of the multiprocess model used in Google’s Chrome can be found at <http://blog.chromium.org/2008/09/multi-process-architecture.html>.

Message passing for multicore systems is discussed in [Holland and Seltzer (2011)]. [Levin (2013)] describes message passing in the Mach system, particularly with respect to macOS and iOS.

[Harold (2005)] provides coverage of socket programming in Java. Details on Android RPCs can be found at <https://developer.android.com/guide/components/aidl.html>. [Hart (2005)] and [Robbins and Robbins (2003)] cover pipes in Windows and UNIX systems, respectively.

Guidelines for Android development can be found at <https://developer.android.com/guide/>.

Bibliography

[Harold (2005)] E. R. Harold, *Java Network Programming*, Third Edition, O’Reilly & Associates (2005).

[Hart (2005)] J. M. Hart, *Windows System Programming*, Third Edition, Addison-Wesley (2005).

- [**Holland and Seltzer (2011)**] D. Holland and M. Seltzer, “Multicore OSes: Looking Forward from 1991, er, 2011”, *Proceedings of the 13th USENIX conference on Hot topics in operating systems* (2011), pages 33–33.
- [**Levin (2013)**] J. Levin, *Mac OS X and iOS Internals to the Apple’s Core*, Wiley (2013).
- [**Love (2010)**] R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).
- [**Robbins and Robbins (2003)**] K. Robbins and S. Robbins, *Unix Systems Programming: Communication, Concurrency and Threads*, Second Edition, Prentice Hall (2003).
- [**Russinovich et al. (2017)**] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).

Chapter 3 Exercises



- 3.8 Describe the actions taken by a kernel to context-switch between processes.
- 3.9 Construct a process tree similar to Figure 3.7. To obtain process information for the UNIX or Linux system, use the command `ps -ael`. Use the command `man ps` to get more information about the `ps` command. The task manager on Windows systems does not provide the parent process ID, but the *process monitor* tool, available from technet.microsoft.com, provides a process-tree tool.
- 3.10 Explain the role of the `init` (or `systemd`) process on UNIX and Linux systems in regard to process termination.
- 3.11 Including the initial parent process, how many processes are created by the program shown in Figure 3.32?
- 3.12 Explain the circumstances under which the line of code marked `printf("LINE J")` in Figure 3.33 will be reached.
- 3.13 Using the program in Figure 3.34, identify the values of `pid` at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)
- 3.14 Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.
- 3.15 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the “at most once” or “exactly once” semantic. Describe possible uses for a mechanism that has neither of these guarantees.
- 3.16 Using the program shown in Figure 3.35, explain what the output will be at lines X and Y.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    for (i = 0; i < 4; i++)
        fork();

    return 0;
}
```

Figure 3.21 How many processes are created?

- 3.17 What are the benefits and the disadvantages of each of the following?
Consider both the system level and the programmer level.

- a. Synchronous and asynchronous communication
- b. Automatic and explicit buffering
- c. Send by copy and send by reference
- d. Fixed-sized and variable-sized messages

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL);
    printf("LINE J");
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
}

return 0;
}
```

Figure 3.22 When will LINE J be reached?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    pid1 = getpid();
    printf("child: pid = %d",pid); /* A */
    printf("child: pid1 = %d",pid1); /* B */
}
else { /* parent process */
    pid1 = getpid();
    printf("parent: pid = %d",pid); /* C */
    printf("parent: pid1 = %d",pid1); /* D */
    wait(NULL);
}

return 0;
}
```

Figure 3.23 What are the pid values?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
    int i;
    pid_t pid;

    pid = fork();

    if (pid == 0) {
        for (i = 0; i < SIZE; i++) {
            nums[i] *= -i;
            printf("CHILD: %d ",nums[i]); /* LINE X */
        }
    } else if (pid > 0) {
        wait(NULL);
        for (i = 0; i < SIZE; i++)
            printf("PARENT: %d ",nums[i]); /* LINE Y */
    }

    return 0;
}
```

Figure 3.24 What output will be at Line X and Line Y?

Programming Problems

- 3.18 Using either a UNIX or a Linux system, write a C program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds. Process states can be obtained from the command

```
ps -l
```

The process states are shown below the S column; processes with a state of Z are zombies. The process identifier (pid) of the child process is listed in the PID column, and that of the parent is listed in the PPID column.

Perhaps the easiest way to determine that the child process is indeed a zombie is to run the program that you have written in the background (using the &) and then run the command ps -l to determine whether the child is a zombie process. Because you do not want too many zombie processes existing in the system, you will need to remove the one that you have created. The easiest way to do that is to terminate the parent process using the kill command. For example, if the pid of the parent is 4884, you would enter

```
kill -9 4884
```

- 3.19 Write a C program called time.c that determines the amount of time necessary to run a command from the command line. This program will be run as "./time <command>" and will report the amount of elapsed time to run the specified command. This will involve using fork() and exec() functions, as well as the gettimeofday() function to determine the elapsed time. It will also require the use of two different IPC mechanisms.

The general strategy is to fork a child process that will execute the specified command. However, before the child executes the command, it will record a timestamp of the current time (which we term “starting time”). The parent process will wait for the child process to terminate. Once the child terminates, the parent will record the current timestamp for the ending time. The difference between the starting and ending times represents the elapsed time to execute the command. The example output below reports the amount of time to run the command ls :

```
./time ls  
time.c  
time
```

```
Elapsed time: 0.25422
```

As the parent and child are separate processes, they will need to arrange how the starting time will be shared between them. You will write two versions of this program, each representing a different method of IPC.

The first version will have the child process write the starting time to a region of shared memory before it calls `exec()`. After the child process terminates, the parent will read the starting time from shared memory. Refer to Section 3.7.1 for details using POSIX shared memory. In that section, there are separate programs for the producer and consumer. As the solution to this problem requires only a single program, the region of shared memory can be established before the child process is forked, allowing both the parent and child processes access to the region of shared memory.

The second version will use a pipe. The child will write the starting time to the pipe, and the parent will read from it following the termination of the child process.

You will use the `gettimeofday()` function to record the current timestamp. This function is passed a pointer to a `struct timeval` object, which contains two members: `tv_sec` and `t_usec`. These represent the number of elapsed seconds and microseconds since January 1, 1970 (known as the UNIX EPOCH). The following code sample illustrates how this function can be used:

```
struct timeval current;

gettimeofday(&current,NULL);

// current.tv_sec represents seconds
// current.tv_usec represents microseconds
```

For IPC between the child and parent processes, the contents of the shared memory pointer can be assigned the `struct timeval` representing the starting time. When pipes are used, a pointer to a `struct timeval` can be written to—and read from—the pipe.

- 3.20 An operating system's **pid manager** is responsible for managing process identifiers. When a process is first created, it is assigned a unique pid by the pid manager. The pid is returned to the pid manager when the process completes execution, and the manager may later reassign this pid. Process identifiers are discussed more fully in Section 3.3.1. What is most important here is to recognize that process identifiers must be unique; no two active processes can have the same pid.

Use the following constants to identify the range of possible pid values:

```
#define MIN_PID 300
#define MAX_PID 5000
```

You may use any data structure of your choice to represent the availability of process identifiers. One strategy is to adopt what Linux has done and use a bitmap in which a value of 0 at position i indicates that

a process id of value i is available and a value of 1 indicates that the process id is currently in use.

Implement the following API for obtaining and releasing a pid:

- `int allocate_map(void)`—Creates and initializes a data structure for representing pids; returns -1 if unsuccessful, 1 if successful
- `int allocate_pid(void)`—Allocates and returns a pid; returns -1 if unable to allocate a pid (all pids are in use)
- `void release_pid(int pid)`—Releases a pid

This programming problem will be modified later on in Chapter 4 and in Chapter 6.

- 3.21** The Collatz conjecture concerns what happens when we take any positive integer n and apply the following algorithm:

$$n = \begin{cases} n/2, & \text{if } n \text{ is even} \\ 3 \times n + 1, & \text{if } n \text{ is odd} \end{cases}$$

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if $n = 35$, the sequence is

35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Write a C program using the `fork()` system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output 8,4,2,1. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

- 3.22** In Exercise 3.21, the child process must output the sequence of numbers generated from the algorithm specified by the Collatz conjecture because the parent and child have their own copies of the data. Another approach to designing this program is to establish a shared-memory object between the parent and child processes. This technique allows the child to write the contents of the sequence to the shared-memory object. The parent can then output the sequence when the child completes. Because the memory is shared, any changes the child makes will be reflected in the parent process as well.

This program will be structured using POSIX shared memory as described in Section 3.7.1. The parent process will progress through the following steps:

- a. Establish the shared-memory object (`shm_open()`, `ftruncate()`, and `mmap()`).

- b. Create the child process and wait for it to terminate.
- c. Output the contents of shared memory.
- d. Remove the shared-memory object.

One area of concern with cooperating processes involves synchronization issues. In this exercise, the parent and child processes must be coordinated so that the parent does not output the sequence until the child finishes execution. These two processes will be synchronized using the `wait()` system call: the parent process will invoke `wait()`, which will suspend it until the child process exits.

- 3.23** Section 3.8.1 describes certain port numbers as being well known—that is, they provide standard services. Port 17 is known as the *quote-of-the-day* service. When a client connects to port 17 on a server, the server responds with a quote for that day.

Modify the date server shown in Figure 3.27 so that it delivers a quote of the day rather than the current date. The quotes should be printable ASCII characters and should contain fewer than 512 characters, although multiple lines are allowed. Since these well-known ports are reserved and therefore unavailable, have your server listen to port 6017. The date client shown in Figure 3.28 can be used to read the quotes returned by your server.

- 3.24** A **haiku** is a three-line poem in which the first line contains five syllables, the second line contains seven syllables, and the third line contains five syllables. Write a haiku server that listens to port 5575. When a client connects to this port, the server responds with a haiku. The date client shown in Figure 3.28 can be used to read the quotes returned by your haiku server.

- 3.25** An echo server echoes back whatever it receives from a client. For example, if a client sends the server the string `Hello there!`, the server will respond with `Hello there!`

Write an echo server using the Java networking API described in Section 3.8.1. This server will wait for a client connection using the `accept()` method. When a client connection is received, the server will loop, performing the following steps:

- Read data from the socket into a buffer.
- Write the contents of the buffer back to the client.

The server will break out of the loop only when it has determined that the client has closed the connection.

The date server of Figure 3.27 uses the `java.io.BufferedReader` class. `BufferedReader` extends the `java.io.Reader` class, which is used for reading character streams. However, the echo server cannot guarantee that it will read characters from clients; it may receive binary data as well. The class `java.io.InputStream` deals with data at the byte level rather than the character level. Thus, your echo server must use an object that extends `java.io.InputStream`. The `read()` method in the

`java.io.InputStream` class returns `-1` when the client has closed its end of the socket connection.

- 3.26 Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message `Hi There`, the second process will return `hI tHERE`. This will require using two pipes, one for sending the original message from the first to the second process and the other for sending the modified message from the second to the first process. You can write this program using either UNIX or Windows pipes.
- 3.27 Design a file-copying program named `filecopy.c` using ordinary pipes. This program will be passed two parameters: the name of the file to be copied and the name of the destination file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows:

```
./filecopy input.txt copy.txt
```

the file `input.txt` will be written to the pipe. The child process will read the contents of this file and write it to the destination file `copy.txt`. You may write this program using either UNIX or Windows pipes.

Programming Projects

Project 1—UNIX Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX `fork()`, `exec()`, `wait()`, `dup2()`, and `pipe()` system calls and can be completed on any Linux, UNIX, or macOS system.

I. Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c`. (This command displays the file `prog.c` on the terminal using the UNIX `cat` command.)

```
osh>cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.9. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (`&`) at the end of the command. Thus, if we rewrite the above command as

```
osh>cat prog.c &
```

the parent and child processes will run concurrently.

The separate child process is created using the `fork()` system call, and the user's command is executed using one of the system calls in the `exec()` family (as described in Section 3.3.1).

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. The `main()` function presents the prompt `osh->` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should_run` equals 1; when the user enters `exit` at the prompt, your program will set `should_run` to 0 and terminate.

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
    char *args[MAX_LINE/2 + 1]; /* command line arguments */
    int should_run = 1; /* flag to determine when to exit program */

    while (should_run) {
        printf("osh>");
        fflush(stdout);

        /**
         * After reading user input, the steps are:
         * (1) fork a child process using fork()
         * (2) the child process will invoke execvp()
         * (3) parent will invoke wait() unless command included &
         */
    }

    return 0;
}
```

Figure 3.36 Outline of simple shell.

This project is organized into several parts:

1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection
4. Allowing the parent and child processes to communicate via a pipe

II. Executing Command in a Child Process

The first task is to modify the `main()` function in Figure 3.36 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (`args` in Figure 3.36). For example, if the user enters the command `ps -ael` at the `osh>` prompt, the values stored in the `args` array are:

```
args[0] = "ps"  
args[1] = "-ael"  
args[2] = NULL
```

This `args` array will be passed to the `execvp()` function, which has the following prototype:

```
execvp(char *command, char *params [])
```

Here, `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0], args)`. Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

III. Creating a History Feature

The next task is to modify the shell interface program so that it provides a *history* feature to allow a user to execute the most recent command by entering `!!`. For example, if a user enters the command `ls -l`, she can then execute that command again by entering `!!` at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command.

Your program should also manage basic error handling. If there is no recent command in the history, entering `!!` should result in a message “No commands in history.”

IV. Redirecting Input and Output

Your shell should then be modified to support the '`>`' and '`<`' redirection

operators, where ‘>’ redirects the output of a command to a file and ‘<’ redirects the input to a command from a file. For example, if a user enters

```
osh>ls > out.txt
```

the output from the ls command will be redirected to the file out.txt. Similarly, input can be redirected as well. For example, if the user enters

```
osh>sort < in.txt
```

the file in.txt will serve as input to the sort command.

Managing the redirection of both input and output will involve using the dup2() function, which duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call

```
dup2(fd, STDOUT_FILENO);
```

duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as sort < in.txt > out.txt.

V. Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

```
osh>ls -l | less
```

has the output of the command ls -l serve as the input to the less command. Both the ls and less commands will run as separate processes and will communicate using the UNIX pipe() function described in Section 3.7.4. Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute ls -l). This child will also create another child process (which will execute less) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the dup2() function as described in the previous section. Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.

Project 2 — Linux Kernel Module for Task Information

In this project, you will write a Linux kernel module that uses the /proc file system for displaying a task’s information based on its process identifier value pid. Before beginning this project, be sure you have completed the Linux kernel module programming project in Chapter 2, which involves creating an entry in the /proc file system. This project will involve writing a process identifier to

the file /proc/pid. Once a pid has been written to the /proc file, subsequent reads from /proc/pid will report (1) the command the task is running, (2) the value of the task's pid, and (3) the current state of the task. An example of how your kernel module will be accessed once loaded into the system is as follows:

```
echo "1395" > /proc/pid
cat /proc/pid
command = [bash] pid = [1395] state = [1]
```

The echo command writes the characters "1395" to the /proc/pid file. Your kernel module will read this value and store its integer equivalent as it represents a process identifier. The cat command reads from /proc/pid, where your kernel module will retrieve the three fields from the task_struct associated with the task whose pid value is 1395.

```
ssize_t proc_write(struct file *file, char __user *usr_buf,
       size_t count, loff_t *pos)
{
    int rv = 0;
    char *k_mem;

    /* allocate kernel memory */
    k_mem = kmalloc(count, GFP_KERNEL);

    /* copies user space usr_buf to kernel memory */
    copy_from_user(k_mem, usr_buf, count);

    printk(KERN_INFO "%s\n", k_mem);

    /* return kernel memory */
    kfree(k_mem);

    return count;
}
```

Figure 3.37 The proc_write() function.

I. Writing to the /proc File System

In the kernel module project in Chapter 2, you learned how to read from the /proc file system. We now cover how to write to /proc. Setting the field .write in struct file_operations to

```
.write = proc_write
```

causes the proc_write() function of Figure 3.37 to be called when a write operation is made to /proc/pid

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. The `GFP_KERNEL` flag indicates routine kernel memory allocation. The `copy_from_user()` function copies the contents of `usr_buf` (which contains what has been written to `/proc/pid`) to the recently allocated kernel memory. Your kernel module will have to obtain the integer equivalent of this value using the kernel function `kstrtol()`, which has the signature

```
int kstrtol(const char *str, unsigned int base, long *res)
```

This stores the character equivalent of `str`, which is expressed as a `base` into `res`.

Finally, note that we return memory that was previously allocated with `kmalloc()` back to the kernel with the call to `kfree()`. Careful memory management—which includes releasing memory to prevent *memory leaks*—is crucial when developing kernel-level code.

II. Reading from the /proc File System

Once the process identifier has been stored, any reads from `/proc/pid` will return the name of the command, its process identifier, and its state. As illustrated in Section 3.1, the PCB in Linux is represented by the structure `task_struct`, which is found in the `<linux/sched.h>` include file. Given a process identifier, the function `pid_task()` returns the associated `task_struct`. The signature of this function appears as follows:

```
struct task_struct pid_task(struct pid *pid,
    enum pid_type type)
```

The kernel function `find_vpid(int pid)` can be used to obtain the `struct pid`, and `PIDTYPE_PID` can be used as the `pid_type`.

For a valid `pid` in the system, `pid_task` will return its `task_struct`. You can then display the values of the command, `pid`, and state. (You will probably have to read through the `task_struct` structure in `<linux/sched.h>` to obtain the names of these fields.)

If `pid_task()` is not passed a valid `pid`, it returns `NULL`. Be sure to perform appropriate error checking to check for this condition. If this situation occurs, the kernel module function associated with reading from `/proc/pid` should return 0.

In the source code download, we give the C program `pid.c`, which provides some of the basic building blocks for beginning this project.

Project 3—Linux Kernel Module for Listing Tasks

In this project, you will write a kernel module that lists all current tasks in a Linux system. You will iterate through the tasks both linearly and depth first.

Part I—Iterating over Tasks Linearly

In the Linux kernel, the `for_each_process()` macro easily allows iteration over all current tasks in the system:

```
#include <linux/sched.h>

struct task_struct *task;

for_each_process(task) {
    /* on each iteration task points to the next task */
}
```

The various fields in `task_struct` can then be displayed as the program loops through the `for_each_process()` macro.

Assignment

Design a kernel module that iterates through all tasks in the system using the `for_each_process()` macro. In particular, output the task command, state, and process id of each task. (You will probably have to read through the `task_struct` structure in `<linux/sched.h>` to obtain the names of these fields.) Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the `dmesg` command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all tasks in the system:

```
ps -el
```

The two values should be very similar. Because tasks are dynamic, however, it is possible that a few tasks may appear in one listing but not the other.

Part II—Iterating over Tasks with a Depth-First Search Tree

The second portion of this project involves iterating over all tasks in the system using a depth-first search (DFS) tree. (As an example: the DFS iteration of the processes in Figure 3.7 is 1, 8415, 8416, 9298, 9204, 2808, 3028, 3610, 4005.)

Linux maintains its process tree as a series of lists. Examining the `task_struct` in `<linux/sched.h>`, we see two `struct list_head` objects:

`children`

and

`sibling`

These objects are pointers to a list of the task's children, as well as its siblings. Linux also maintains a reference to the initial task in the system — `init_task` — which is of type `task_struct`. Using this information as well as macro operations on lists, we can iterate over the children of `init_task` as follows:

```
struct task_struct *task;
struct list_head *list;
```

```

list_for_each(list, &init_task->children) {
    task = list_entry(list, struct task_struct, sibling);
    /* task points to the next child in the list */
}

```

The `list_for_each()` macro is passed two parameters, both of type `struct list_head`:

- A pointer to the head of the list to be traversed
- A pointer to the head node of the list to be traversed

At each iteration of `list_for_each()`, the first parameter is set to the `list` structure of the next child. We then use this value to obtain each structure in the list using the `list_entry()` macro.

Assignment

Beginning from `init_task` task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, output the name, state, and pid of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer.

If you output all tasks in the system, you may see many more tasks than appear with the `ps -ael` command. This is because some threads appear as children but do not show up as ordinary processes. Therefore, to check the output of the DFS tree, use the command

```
ps -eLf
```

This command lists all tasks—including threads—in the system. To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the `ps` command.

Project 4—Kernel Data Structures

In Section 1.9, we covered various data structures that are common in operating systems. The Linux kernel provides several of these structures. Here, we explore using the circular, doubly linked list that is available to kernel developers. Much of what we discuss is available in the Linux source code—in this instance, the include file `<linux/list.h>`—and we recommend that you examine this file as you proceed through the following steps.

Initially, you must define a `struct` containing the elements that are to be inserted in the linked list. The following C `struct` defines a color as a mixture of red, blue, and green:

```

struct color {
    int red;
    int blue;
    int green;
}

```

```
    struct list_head list;
};
```

Notice the member `struct list_head list`. The `list_head` structure is defined in the include file `<linux/types.h>`, and its intention is to embed the linked list within the nodes that comprise the list. This `list_head` structure is quite simple—it merely holds two members, `next` and `prev`, that point to the next and previous entries in the list. By embedding the linked list within the structure, Linux makes it possible to manage the data structure with a series of *macro* functions.

I. Inserting Elements into the Linked List

We can declare a `list_head` object, which we use as a reference to the head of the list by using the `LIST_HEAD()` macro:

```
static LIST_HEAD(color_list);
```

This macro defines and initializes the variable `color_list`, which is of type `struct list_head`.

We create and initialize instances of `struct color` as follows:

```
struct color *violet;

violet = kmalloc(sizeof(*violet), GFP_KERNEL);
violet->red = 138;
violet->blue = 43;
violet->green = 226;
INIT_LIST_HEAD(&violet->list);
```

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. The `GFP_KERNEL` flag indicates routine kernel memory allocation. The macro `INIT_LIST_HEAD()` initializes the `list` member in `struct color`. We can then add this instance to the end of the linked list using the `list_add_tail()` macro:

```
list_add_tail(&violet->list, &color_list);
```

II. Traversing the Linked List

Traversing the list involves using the `list_for_each_entry()` macro, which accepts three parameters:

- A pointer to the structure being iterated over
- A pointer to the head of the list being iterated over
- The name of the variable containing the `list_head` structure

The following code illustrates this macro:

```

struct color *ptr;

list_for_each_entry(ptr, &color_list, list) {
    /* on each iteration ptr points */
    /* to the next struct color */
}

```

III. Removing Elements from the Linked List

Removing elements from the list involves using the `list_del()` macro, which is passed a pointer to `struct list_head`:

```
list_del(struct list_head *element);
```

This removes `element` from the list while maintaining the structure of the remainder of the list.

Perhaps the simplest approach for removing all elements from a linked list is to remove each element as you traverse the list. The macro `list_for_each_entry_safe()` behaves much like `list_for_each_entry()` except that it is passed an additional argument that maintains the value of the `next` pointer of the item being deleted. (This is necessary for preserving the structure of the list.) The following code example illustrates this macro:

```

struct color *ptr, *next;

list_for_each_entry_safe(ptr,next,&color_list,list) {
    /* on each iteration ptr points */
    /* to the next struct color */
    list_del(&ptr->list);
    kfree(ptr);
}

```

Notice that after deleting each element, we return memory that was previously allocated with `kmalloc()` back to the kernel with the call to `kfree()`.

Part I—Assignment

In the module entry point, create a linked list containing four `struct color` elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the `dmesg` command to ensure that the list is properly constructed once the kernel module has been loaded.

In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the `dmesg` command to check that the list has been removed once the kernel module has been unloaded.

Part II—Parameter Passing

This portion of the project will involve passing a parameter to a kernel module. The module will use this parameter as an initial value and generate the Collatz sequence as described in Exercise 3.21.

Passing a Parameter to a Kernel Module

Parameters may be passed to kernel modules when they are loaded. For example, if the name of the kernel module is `collatz`, we can pass the initial value of 15 to the kernel parameter `start` as follows:

```
sudo insmod collatz.ko start=15
```

Within the kernel module, we declare `start` as a parameter using the following code:

```
#include<linux/moduleparam.h>

static int start = 25;

module_param(start, int, 0);
```

The `module_param()` macro is used to establish variables as parameters to kernel modules. `module_param()` is provided three arguments: (1) the name of the parameter, (2) its type, and (3) file permissions. Since we are not using a file system for accessing the parameter, we are not concerned with permissions and use a default value of 0. Note that the name of the parameter used with the `insmod` command must match the name of the associated kernel parameter. Finally, if we do not provide a value to the module parameter during loading with `insmod`, the default value (which in this case is 25) is used.

Part II—Assignment

Design a kernel module named `collatz` that is passed an initial value as a module parameter. Your module will then generate and store the sequence in a kernel linked list when the module is loaded. Once the sequence has been stored, your module will traverse the list and output its contents to the kernel log buffer. Use the `dmesg` command to ensure that the sequence is properly generated once the module has been loaded.

In the module exit point, delete the contents of the list and return the free memory back to the kernel. Again, use `dmesg` to check that the list has been removed once the kernel module has been unloaded.

Threads & Concurrency



The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control. Identifying opportunities for parallelism through the use of threads is becoming increasingly important for modern multicore systems that provide multiple CPUs.

In this chapter, we introduce many concepts, as well as challenges, associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Windows, and Java thread libraries. Additionally, we explore several new features that abstract the concept of creating threads, allowing developers to focus on identifying opportunities for parallelism and letting language features and API frameworks manage the details of thread creation and management. We look at a number of issues related to multithreaded programming and its effect on the design of operating systems. Finally, we explore how the Windows and Linux operating systems support threads at the kernel level.

CHAPTER OBJECTIVES

- Identify the basic components of a thread, and contrast threads and processes.
- Describe the major benefits and significant challenges of designing multi-threaded processes.
- Illustrate different approaches to implicit threading, including thread pools, fork-join, and Grand Central Dispatch.
- Describe how the Windows and Linux operating systems represent threads.
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs.

4.1 Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 4.1 illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

4.1.1 Motivation

Most software applications that run on modern computers and mobile devices are multithreaded. An application typically is implemented as a separate process with several threads of control. Below we highlight a few examples of multithreaded applications:

- An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
- A web browser might have one thread display images or text while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

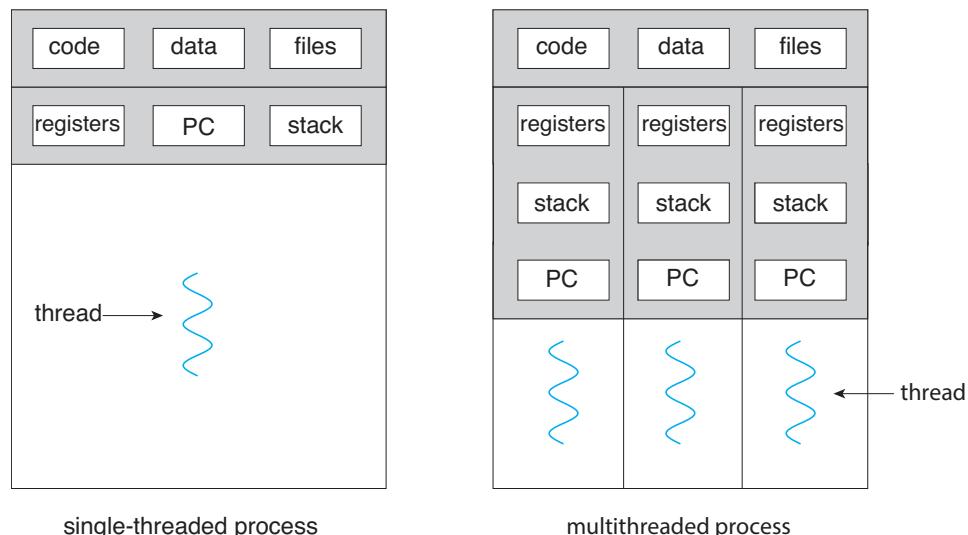


Figure 4.1 Single-threaded and multithreaded processes.

Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resumes listening for additional requests. This is illustrated in Figure 4.2.

Most operating system kernels are also typically multithreaded. As an example, during system boot time on Linux systems, several kernel threads are created. Each thread performs a specific task, such as managing devices, memory management, or interrupt handling. The command `ps -ef` can be used to display the kernel threads on a running Linux system. Examining the output of this command will show the kernel thread `kthreadd` (with `pid = 2`), which serves as the parent of all other kernel threads.

Many applications can also take advantage of multiple threads, including basic sorting, trees, and graph algorithms. In addition, programmers who must solve contemporary CPU-intensive problems in data mining, graphics, and artificial intelligence can leverage the power of modern multicore systems by designing solutions that run in parallel.

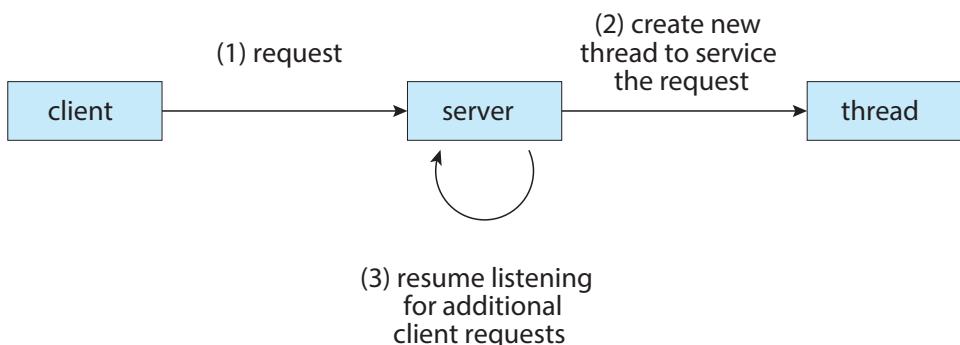


Figure 4.2 Multithreaded server architecture.

4.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness.** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
2. **Resource sharing.** Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy.** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

4.2 Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A later, yet similar, trend in system design is to place multiple computing cores on a single processing chip where each core appears as a separate CPU to the operating system (Section 1.3.2). We refer to such systems as **multicore**, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency

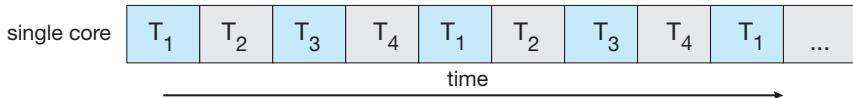


Figure 4.3 Concurrent execution on a single-core system.

means that some threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

Notice the distinction between *concurrency* and *parallelism* in this discussion. A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism. Before the advent of multiprocessor and multicore architectures, most computer systems had only a single processor, and CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

4.2.1 Programming Challenges

The trend toward multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple computing cores. Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution shown in Figure 4.4. For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.

In general, five areas present challenges in programming for multicore systems:

1. **Identifying tasks.** This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
2. **Balance.** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

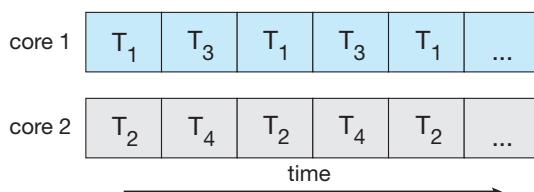


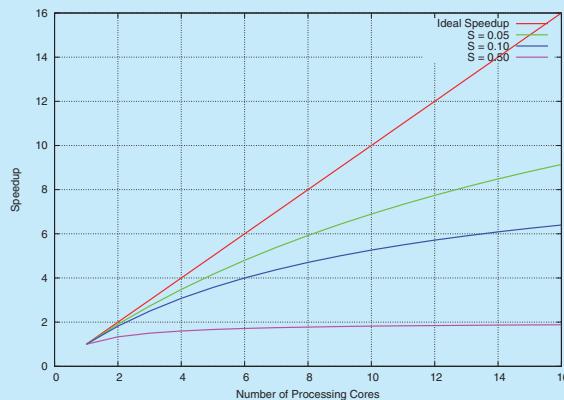
Figure 4.4 Parallel execution on a multicore system.

AMDAHL'S LAW

Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add two additional cores (for a total of four), the speedup is 2.28 times. Below is a graph illustrating Amdahl's Law in several different scenarios.



One interesting fact about Amdahl's Law is that as N approaches infinity, the speedup converges to $1/S$. For example, if 50 percent of an application is performed serially, the maximum speedup is 2.0 times, regardless of the number of processing cores we add. This is the fundamental principle behind Amdahl's Law: the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

3. **Data splitting.** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency.** The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency. We examine such strategies in Chapter 6.

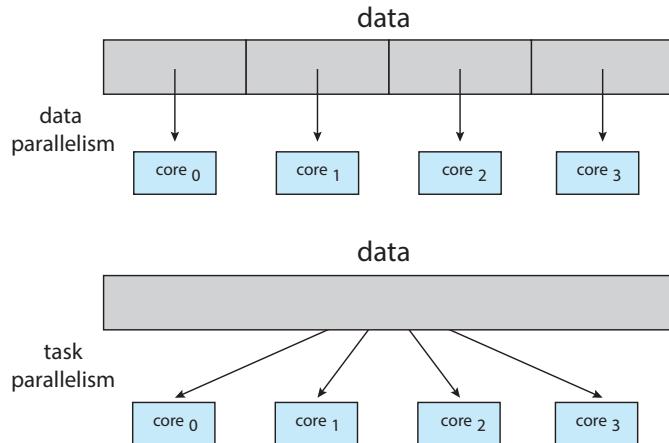


Figure 4.5 Data and task parallelism.

5. **Testing and debugging.** When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

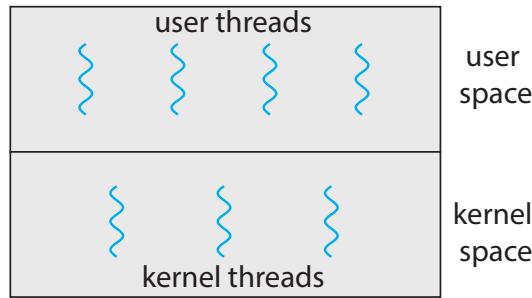
Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future. (Similarly, many computer science educators believe that software development must be taught with increased emphasis on parallel programming.)

4.2.2 Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Consider, for example, summing the contents of an array of size N . On a single-core system, one thread would simply sum the elements $[0] \dots [N - 1]$. On a dual-core system, however, thread A , running on core 0, could sum the elements $[0] \dots [N/2 - 1]$ while thread B , running on core 1, could sum the elements $[N/2] \dots [N - 1]$. The two threads would be running in parallel on separate computing cores.

Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data. Consider again our example above. In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

Fundamentally, then, data parallelism involves the distribution of data across multiple cores, and task parallelism involves the distribution of tasks across multiple cores, as shown in Figure 4.5. However, data and task paral-

**Figure 4.6** User and kernel threads.

lism are not mutually exclusive, and an application may in fact use a hybrid of these two strategies.

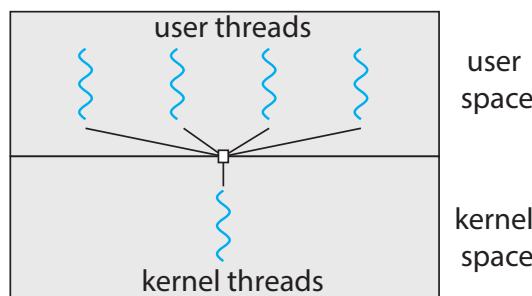
4.3 Multithreading Models

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, and macOS—support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads, as illustrated in Figure 4.6. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

4.3.1 Many-to-One Model

The many-to-one model (Figure 4.7) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient (we discuss thread libraries in Section 4.4). However, the entire process will block if a thread makes a blocking system call. Also, because only

**Figure 4.7** Many-to-one model.

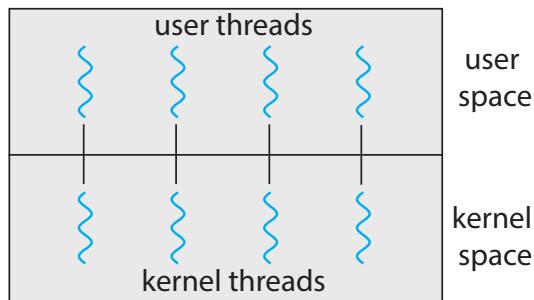


Figure 4.8 One-to-one model.

one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model. However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores, which have now become standard on most computer systems.

4.3.2 One-to-One Model

The one-to-one model (Figure 4.8) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

4.3.3 Many-to-Many Model

The many-to-many model (Figure 4.9) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores).

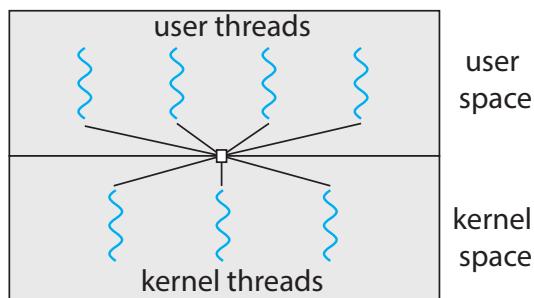


Figure 4.9 Many-to-many model.

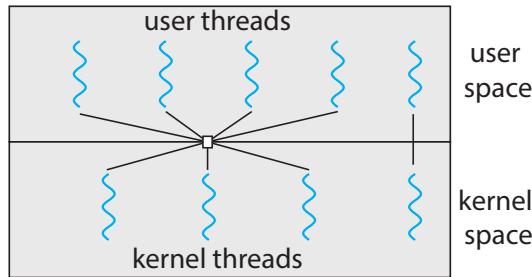


Figure 4.10 Two-level model.

Let's consider the effect of this design on concurrency. Whereas the many-to-one model allows the developer to create as many user threads as she wishes, it does not result in parallelism, because the kernel can schedule only one kernel thread at a time. The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application. (In fact, on some systems, she may be limited in the number of threads she can create.) The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model** (Figure 4.10).

Although the many-to-many model appears to be the most flexible of the models discussed, in practice it is difficult to implement. In addition, with an increasing number of processing cores appearing on most systems, limiting the number of kernel threads has become less important. As a result, most operating systems now use the one-to-one model. However, as we shall see in Section 4.5, some contemporary concurrency libraries have developers identify tasks that are then mapped to threads using the many-to-many model.

4.4 Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java. Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library. The Windows thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX, Linux, and macOS systems typically use Pthreads.

For POSIX and Windows threading, any data declared globally—that is, declared outside of any function—are shared among all threads belonging to the same process. Because Java has no equivalent notion of global data, access to shared data must be explicitly arranged between threads.

In the remainder of this section, we describe basic thread creation using these three thread libraries. As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$\text{sum} = \sum_{i=1}^N i$$

For example, if N were 5, this function would represent the summation of integers from 1 to 5, which is 15. Each of the three programs will be run with the upper bounds of the summation entered on the command line. Thus, if the user enters 8, the summation of the integer values from 1 to 8 will be output.

Before we proceed with our examples of thread creation, we introduce two general strategies for creating multiple threads: **asynchronous threading** and **synchronous threading**. With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another. Because the threads are independent, there is typically little data sharing between them. Asynchronous threading is the strategy used in the multithreaded server illustrated in Figure 4.2 and is also commonly used for designing responsive user interfaces.

Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes. Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed. Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution. Typically, synchronous threading involves significant data sharing among threads. For example, the parent thread may combine the results calculated by its various children. All of the following examples use synchronous threading.

4.4.1 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a *specification* for thread behavior, not an *implementation*. Operating-system designers may implement the specification

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Figure 4.11 Multithreaded C program using the Pthreads API.

in any way they wish. Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux and macOS. Although Windows doesn't support Pthreads natively, some third-party implementations for Windows are available.

The C program shown in Figure 4.11 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread. In a Pthreads program, separate threads begin execution in a specified function. In Figure 4.11, this is the `runner()` function. When this program begins, a single thread of control begins in

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Figure 4.12 Pthread code for joining ten threads.

`main()`. After some initialization, `main()` creates a second thread that begins control in the `runner()` function. Both threads share the global data `sum`.

Let's look more closely at this program. All Pthreads programs must include the `pthread.h` header file. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The `pthread_attr_t attr` declaration represents the attributes for the thread. We set the attributes in the function call `pthread_attr_init(&attr)`. Because we did not explicitly set any attributes, we use the default attributes provided. (In Chapter 5, we discuss some of the scheduling attributes provided by the Pthreads API.) A separate thread is created with the `pthread_create()` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the `runner()` function. Last, we pass the integer parameter that was provided on the command line, `argv[1]`.

At this point, the program has two threads: the initial (or parent) thread in `main()` and the summation (or child) thread performing the summation operation in the `runner()` function. This program follows the thread create/join strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the `pthread_join()` function. The summation thread will terminate when it calls the function `pthread_exit()`. Once the summation thread has returned, the parent thread will output the value of the shared data `sum`.

This example program creates only a single thread. With the growing dominance of multicore systems, writing programs containing several threads has become increasingly common. A simple method for waiting on several threads using the `pthread_join()` function is to enclose the operation within a simple `for` loop. For example, you can join on ten threads using the Pthread code shown in Figure 4.12.

4.4.2 Windows Threads

The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways. We illustrate the Windows thread API in the C program shown in Figure 4.13. Notice that we must include the `windows.h` header file when using the Windows API.

```

#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}

```

Figure 4.13 Multithreaded C program using the Windows API.

Just as in the Pthreads version shown in Figure 4.11, data shared by the separate threads—in this case, Sum—are declared globally (the DWORD data type is an unsigned 32-bit integer). We also define the `Summation()` function that is to be performed in a separate thread. This function is passed a pointer to a void, which Windows defines as LPVOID. The thread performing this function sets the global data Sum to the value of the summation from 0 to the parameter passed to `Summation()`.

Threads are created in the Windows API using the `CreateThread()` function, and—just as in Pthreads—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes. (The default values do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler.) Once the summation thread is created, the parent must wait for it to complete before outputting the value of `Sum`, as the value is set by the summation thread. Recall that the Pthread program (Figure 4.11) had the parent thread wait for the summation thread using the `pthread_join()` statement. We perform the equivalent of this in the Windows API using the `WaitForSingleObject()` function, which causes the creating thread to block until the summation thread has exited.

In situations that require waiting for multiple threads to complete, the `WaitForMultipleObjects()` function is used. This function is passed four parameters:

1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicating whether all objects have been signaled
4. A timeout duration (or `INFINITE`)

For example, if `THandles` is an array of thread `HANDLE` objects of size `N`, the parent thread can wait for all its child threads to complete with this statement:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

4.4.3 Java Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a `main()` method runs as a single thread in the JVM. Java threads are available on any system that provides a JVM including Windows, Linux, and macOS. The Java thread API is available for Android applications as well.

There are two techniques for explicitly creating threads in a Java program. One approach is to create a new class that is derived from the `Thread` class and to override its `run()` method. An alternative—and more commonly used—technique is to define a class that implements the `Runnable` interface. This interface defines a single abstract method with the signature `public void run()`. The code in the `run()` method of a class that implements `Runnable` is what executes in a separate thread. An example is shown below:

```
class Task implements Runnable
{
    public void run() {
        System.out.println("I am a thread.");
    }
}
```

LAMBDA EXPRESSIONS IN JAVA

Beginning with Version 1.8 of the language, Java introduced Lambda expressions, which allow a much cleaner syntax for creating threads. Rather than defining a separate class that implements `Runnable`, a Lambda expression can be used instead:

```
Runnable task = () -> {
    System.out.println("I am a thread.");
};

Thread worker = new Thread(task);
worker.start();
```

Lambda expressions—as well as similar functions known as [closures](#)—are a prominent feature of functional programming languages and have been available in several nonfunctional languages as well including Python, C++, and C#. As we shall see in later examples in this chapter, Lambda expressions often provide a simple syntax for developing parallel applications.

Thread creation in Java involves creating a `Thread` object and passing it an instance of a class that implements `Runnable`, followed by invoking the `start()` method on the `Thread` object. This appears in the following example:

```
Thread worker = new Thread(new Task());
worker.start();
```

Invoking the `start()` method for the new `Thread` object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the `run()` method, making the thread eligible to be run by the JVM. (Note again that we never call the `run()` method directly. Rather, we call the `start()` method, and it calls the `run()` method on our behalf.)

Recall that the parent threads in the Pthreads and Windows libraries use `pthread_join()` and `WaitForSingleObject()` (respectively) to wait for the summation threads to finish before proceeding. The `join()` method in Java provides similar functionality. (Notice that `join()` can throw an `InterruptedException`, which we choose to ignore.)

```
try {
    worker.join();
}
catch (InterruptedException ie) { }
```

If the parent must wait for several threads to finish, the `join()` method can be enclosed in a `for` loop similar to that shown for Pthreads in Figure 4.12.

4.4.3.1 Java Executor Framework

Java has supported thread creation using the approach we have described thus far since its origins. However, beginning with Version 1.5 and its API, Java introduced several new concurrency features that provide developers with much greater control over thread creation and communication. These tools are available in the `java.util.concurrent` package.

Rather than explicitly creating `Thread` objects, thread creation is instead organized around the `Executor` interface:

```
public interface Executor
{
    void execute(Runnable command);
}
```

Classes implementing this interface must define the `execute()` method, which is passed a `Runnable` object. For Java developers, this means using the `Executor` rather than creating a separate `Thread` object and invoking its `start()` method. The `Executor` is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```

The `Executor` framework is based on the producer-consumer model; tasks implementing the `Runnable` interface are produced, and the threads that execute these tasks consume them. The advantage of this approach is that it not only divides thread creation from execution but also provides a mechanism for communication between concurrent tasks.

Data sharing between threads belonging to the same process occurs easily in Windows and Pthreads, since shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data. We can pass parameters to a class that implements `Runnable`, but Java threads cannot return results. To address this need, the `java.util.concurrent` package additionally defines the `Callable` interface, which behaves similarly to `Runnable` except that a result can be returned. Results returned from `Callable` tasks are known as `Future` objects. A result can be retrieved from the `get()` method defined in the `Future` interface. The program shown in Figure 4.14 illustrates the summation program using these Java features.

The `Summation` class implements the `Callable` interface, which specifies the method `V call()`—it is the code in this `call()` method that is executed in a separate thread. To execute this code, we create a `newSingleThreadExecutor` object (provided as a static method in the `Executors` class), which is of type `ExecutorService`, and pass it a `Callable` task using its `submit()` method. (The primary difference between the `execute()` and `submit()` methods is that the former returns no result, whereas the latter returns a result as a `Future`.) Once we submit the `callable` task to the thread, we wait for its result by calling the `get()` method of the `Future` object it returns.

It is quite easy to notice at first that this model of thread creation appears more complicated than simply creating a thread and joining on its termination. However, incurring this modest degree of complication confers benefits. As we have seen, using `Callable` and `Future` allows for threads to return results.

```

import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
    private int upper;
    public Summation(int upper) {
        this.upper = upper;
    }

    /* The thread will execute in this method */
    public Integer call() {
        int sum = 0;
        for (int i = 1; i <= upper; i++)
            sum += i;

        return new Integer(sum);
    }
}

public class Driver
{
    public static void main(String[] args) {
        int upper = Integer.parseInt(args[0]);

        ExecutorService pool = Executors.newSingleThreadExecutor();
        Future<Integer> result = pool.submit(new Summation(upper));

        try {
            System.out.println("sum = " + result.get());
        } catch (InterruptedException | ExecutionException ie) { }
    }
}

```

Figure 4.14 Illustration of Java Executor framework API.

Additionally, this approach separates the creation of threads from the results they produce: rather than waiting for a thread to terminate before retrieving results, the parent instead only waits for the results to become available. Finally, as we shall see in Section 4.5.1, this framework can be combined with other features to create robust tools for managing a large number of threads.

4.5 Implicit Threading

With the continued growth of multicore processing, applications containing hundreds—or even thousands—of threads are looming on the horizon. Designing such applications is not a trivial undertaking: programmers must

THE JVM AND THE HOST OPERATING SYSTEM

The JVM is typically implemented on top of a host operating system (see Figure 18.10). This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying operating system, instead leaving that decision to the particular implementation of the JVM. For example, the Windows operating system uses the one-to-one model; therefore, each Java thread for a JVM running on Windows maps to a kernel thread. In addition, there may be a relationship between the Java thread library and the thread library on the host operating system. For example, implementations of a JVM for the Windows family of operating systems might use the Windows API when creating Java threads; Linux and macOS systems might use the Pthreads API.

address not only the challenges outlined in Section 4.2 but additional difficulties as well. These difficulties, which relate to program correctness, are covered in Chapter 6 and Chapter 8.

One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed **implicit threading**, is an increasingly popular trend. In this section, we explore four alternative approaches to designing applications that can take advantage of multicore processors through implicit threading. As we shall see, these strategies generally require application developers to identify *tasks*—not threads—that can run in parallel. A task is usually written as a function, which the run-time library then maps to a separate thread, typically using the many-to-many model (Section 4.3.3). The advantage of this approach is that developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management.

4.5.1 Thread Pools

In Section 4.1, we described a multithreaded web server. In this situation, whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems. The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work. The second issue is more troublesome. If we allow each concurrent request to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.

ANDROID THREAD POOLS

In Section 3.8.2.1, we covered RPCs in the Android operating system. You may recall from that section that Android uses the Android Interface Definition Language (AIDL), a tool that specifies the remote interface that clients interact with on the server. AIDL also provides a thread pool. A remote service using the thread pool can handle multiple concurrent requests, servicing each request using a separate thread from the pool.

The general idea behind a thread pool is to create a number of threads at start-up and place them into a pool, where they sit and wait for work. When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests. If there is an available thread in the pool, it is awakened, and the request is serviced immediately. If the pool contains no available thread, the task is queued until one becomes free. Once a thread completes its service, it returns to the pool and awaits more work. Thread pools work well when the tasks submitted to the pool can be executed asynchronously.

Thread pools offer these benefits:

1. Servicing a request with an existing thread is often faster than waiting to create a thread.
2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further benefit of having a smaller pool—thereby consuming less memory—when the load on the system is low. We discuss one such architecture, Apple’s Grand Central Dispatch, later in this section.

The Windows API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the `Thread_Create()` function, as described in Section 4.4.2. Here, a function that is to run as a separate thread is defined. Such a function may appear as follows:

```
DWORD WINAPI PoolFunction(PVOID Param) {  
    /* this function runs as a separate thread. */  
}
```

A pointer to `PoolFunction()` is passed to one of the functions in the thread pool API, and a thread from the pool executes this function. One such member in the thread pool API is the `QueueUserWorkItem()` function, which is passed three parameters:

- `LPTHREAD_START_ROUTINE Function`—a pointer to the function that is to run as a separate thread
- `PVOID Param`—the parameter passed to `Function`
- `ULONG Flags`—flags indicating how the thread pool is to create and manage execution of the thread

An example of invoking a function is the following:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

This causes a thread from the thread pool to invoke `PoolFunction()` on behalf of the programmer. In this instance, we pass no parameters to `PoolFunction()`. Because we specify 0 as a flag, we provide the thread pool with no special instructions for thread creation.

Other members in the Windows thread pool API include utilities that invoke functions at periodic intervals or when an asynchronous I/O request completes.

4.5.1.1 Java Thread Pools

The `java.util.concurrent` package includes an API for several varieties of thread-pool architectures. Here, we focus on the following three models:

1. Single thread executor—`newSingleThreadExecutor()`—creates a pool of size 1.
2. Fixed thread executor—`newFixedThreadPool(int size)`—creates a thread pool with a specified number of threads.
3. Cached thread executor—`newCachedThreadPool()`—creates an unbounded thread pool, reusing threads in many instances.

We have, in fact, already seen the use of a Java thread pool in Section 4.4.3, where we created a `newSingleThreadExecutor` in the program example shown in Figure 4.14. In that section, we noted that the Java executor framework can be used to construct more robust threading tools. We now describe how it can be used to create thread pools.

A thread pool is created using one of the factory methods in the `Executors` class:

- `static ExecutorService newSingleThreadExecutor()`
- `static ExecutorService newFixedThreadPool(int size)`
- `static ExecutorService newCachedThreadPool()`

Each of these factory methods creates and returns an object instance that implements the `ExecutorService` interface. `ExecutorService` extends the `Exe-`

```

import java.util.concurrent.*;

public class ThreadPoolExample
{
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());

        /* Create the thread pool */
        ExecutorService pool = Executors.newCachedThreadPool();

        /* Run each task using a thread in the pool */
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());

        /* Shut down the pool once all threads have completed */
        pool.shutdown();
    }
}

```

Figure 4.15 Creating a thread pool in Java.

tor interface, allowing us to invoke the `execute()` method on this object. In addition, `ExecutorService` provides methods for managing termination of the thread pool.

The example shown in Figure 4.15 creates a cached thread pool and submits tasks to be executed by a thread in the pool using the `execute()` method. When the `shutdown()` method is invoked, the thread pool rejects additional tasks and shuts down once all existing tasks have completed execution.

4.5.2 Fork Join

The strategy for thread creation covered in Section 4.4 is often known as the **fork-join** model. Recall that with this method, the main parent thread creates (*forks*) one or more child threads and then waits for the children to terminate and *join* with it, at which point it can retrieve and combine their results. This synchronous model is often characterized as explicit thread creation, but it is also an excellent candidate for implicit threading. In the latter situation, threads are not constructed directly during the fork stage; rather, parallel tasks are designated. This model is illustrated in Figure 4.16. A library manages the number of threads that are created and is also responsible for assigning tasks to threads. In some ways, this fork-join model is a synchronous version of thread pools in which a library determines the actual number of threads to create—for example, by using the heuristics described in Section 4.5.1.

4.5.2.1 Fork Join in Java

Java introduced a fork-join library in Version 1.7 of the API that is designed to be used with recursive divide-and-conquer algorithms such as Quicksort and Mergesort. When implementing divide-and-conquer algorithms using this

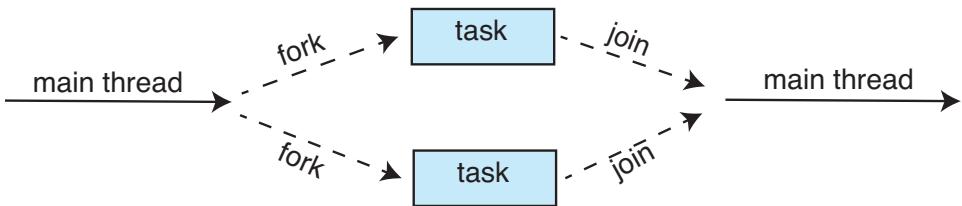


Figure 4.16 Fork-join parallelism.

library, separate tasks are forked during the divide step and assigned smaller subsets of the original problem. Algorithms must be designed so that these separate tasks can execute concurrently. At some point, the size of the problem assigned to a task is small enough that it can be solved directly and requires creating no additional tasks. The general recursive algorithm behind Java's fork-join model is shown below:

```

Task(problem)
  if problem is small enough
    solve the problem directly
  else
    subtask1 = fork(new Task(subset of problem))
    subtask2 = fork(new Task(subset of problem))

    result1 = join(subtask1)
    result2 = join(subtask2)

  return combined results
  
```

Figure 4.17 depicts the model graphically.

We now illustrate Java's fork-join strategy by designing a divide-and-conquer algorithm that sums all elements in an array of integers. In Version 1.7 of the API Java introduced a new thread pool—the `ForkJoinPool`—that can be assigned tasks that inherit the abstract base class `ForkJoinTask` (which for now we will assume is the `SumTask` class). The following creates a `ForkJoinPool` object and submits the initial task via its `invoke()` method:

```

ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
  
```

Upon completion, the initial call to `invoke()` returns the summation of `array`.

The class `SumTask`—shown in Figure 4.18—implements a divide-and-conquer algorithm that sums the contents of the array using fork-join. New tasks are created using the `fork()` method, and the `compute()` method specifies the computation that is performed by each task. The method `compute()` is invoked until it can directly calculate the sum of the subset it is assigned. The

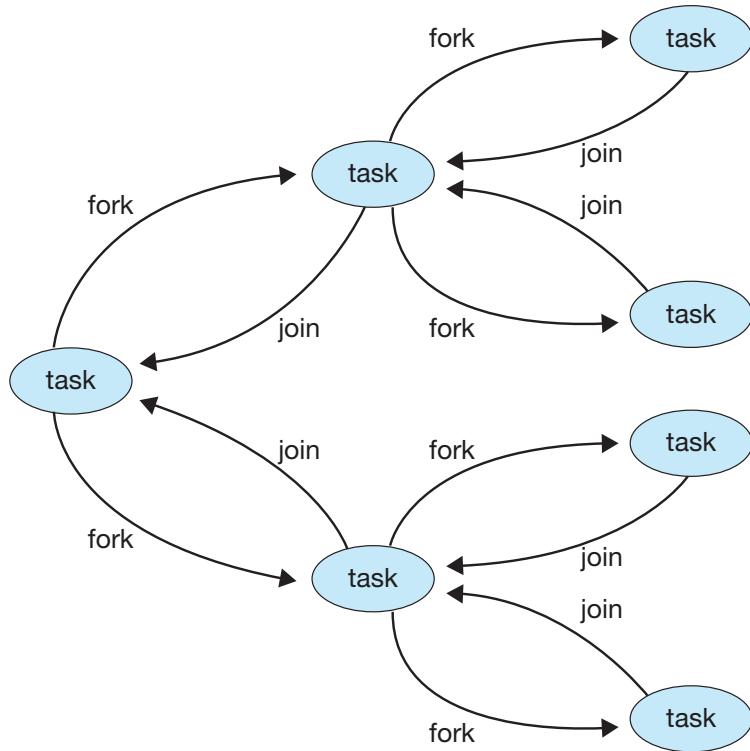


Figure 4.17 Fork-join in Java.

call to `join()` blocks until the task completes, upon which `join()` returns the results calculated in `compute()`.

Notice that `SumTask` in Figure 4.18 extends `RecursiveTask`. The Java fork-join strategy is organized around the abstract base class `ForkJoinTask`, and the `RecursiveTask` and `RecursiveAction` classes extend this class. The fundamental difference between these two classes is that `RecursiveTask` returns a result (via the return value specified in `compute()`), and `RecursiveAction` does not return a result. The relationship between the three classes is illustrated in the UML class diagram in Figure 4.19.

An important issue to consider is determining when the problem is “small enough” to be solved directly and no longer requires creating additional tasks. In `SumTask`, this occurs when the number of elements being summed is less than the value `THRESHOLD`, which in Figure 4.18 we have arbitrarily set to 1,000. In practice, determining when a problem can be solved directly requires careful timing trials, as the value can vary according to implementation.

What is interesting in Java’s fork-join model is the management of tasks wherein the library constructs a pool of worker threads and balances the load of tasks among the available workers. In some situations, there are thousands of tasks, yet only a handful of threads performing the work (for example, a separate thread for each CPU). Additionally, each thread in a `ForkJoinPool` maintains a queue of tasks that it has forked, and if a thread’s queue is empty, it can steal a task from another thread’s queue using a *work stealing* algorithm, thus balancing the workload of tasks among all threads.

```

import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
    static final int THRESHOLD = 1000;

    private int begin;
    private int end;
    private int[] array;

    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }

    protected Integer compute() {
        if (end - begin < THRESHOLD) {
            int sum = 0;
            for (int i = begin; i <= end; i++)
                sum += array[i];

            return sum;
        }
        else {
            int mid = (begin + end) / 2;

            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);

            leftTask.fork();
            rightTask.fork();

            return rightTask.join() + leftTask.join();
        }
    }
}

```

Figure 4.18 Fork-join calculation using the Java API.

4.5.3 OpenMP

OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-

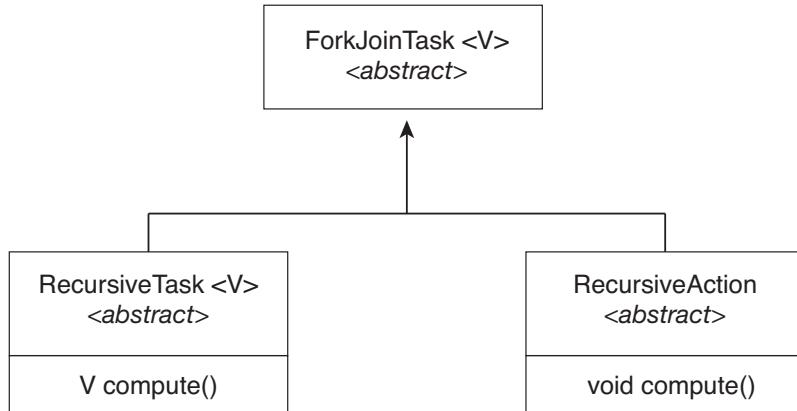


Figure 4.19 UML class diagram for Java’s fork-join.

time library to execute the region in parallel. The following C program illustrates a compiler directive above the parallel region containing the `printf()` statement:

```

#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}

```

When OpenMP encounters the directive

```
#pragma omp parallel
```

it creates as many threads as there are processing cores in the system. Thus, for a dual-core system, two threads are created; for a quad-core system, four are created; and so forth. All the threads then simultaneously execute the parallel region. As each thread exits the parallel region, it is terminated.

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops. For example, assume we have two arrays, `a` and `b`, of size `N`. We wish to sum their contents and place the results

in array `c`. We can have this task run in parallel by using the following code segment, which contains the compiler directive for parallelizing `for` loops:

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

OpenMP divides the work contained in the `for` loop among the threads it has created in response to the directive

```
#pragma omp parallel for
```

In addition to providing directives for parallelization, OpenMP allows developers to choose among several levels of parallelism. For example, they can set the number of threads manually. It also allows developers to identify whether data are shared between threads or are private to a thread. OpenMP is available on several open-source and commercial compilers for Linux, Windows, and macOS systems. We encourage readers interested in learning more about OpenMP to consult the bibliography at the end of the chapter.

4.5.4 Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology developed by Apple for its macOS and iOS operating systems. It is a combination of a run-time library, an API, and language extensions that allow developers to identify sections of code (*tasks*) to run in parallel. Like OpenMP, GCD manages most of the details of threading.

GCD schedules tasks for run-time execution by placing them on a **dispatch queue**. When it removes a task from a queue, it assigns the task to an available thread from a pool of threads that it manages. GCD identifies two types of dispatch queues: *serial* and *concurrent*.

Tasks placed on a serial queue are removed in FIFO order. Once a task has been removed from the queue, it must complete execution before another task is removed. Each process has its own serial queue (known as its **main queue**), and developers can create additional serial queues that are local to a particular process. (This is why serial queues are also known as *private dispatch queues*.) Serial queues are useful for ensuring the sequential execution of several tasks.

Tasks placed on a concurrent queue are also removed in FIFO order, but several tasks may be removed at a time, thus allowing multiple tasks to execute in parallel. There are several system-wide concurrent queues (also known as *global dispatch queues*), which are divided into four primary quality-of-service classes:

- **QOS_CLASS_USER_INTERACTIVE**—The **user-interactive** class represents tasks that interact with the user, such as the user interface and event handling, to ensure a responsive user interface. Completing a task belonging to this class should require only a small amount of work.
- **QOS_CLASS_USER_INITIATED**—The **user-initiated** class is similar to the user-interactive class in that tasks are associated with a responsive user interface; however, user-initiated tasks may require longer processing

times. Opening a file or a URL is a user-initiated task, for example. Tasks belonging to this class must be completed for the user to continue interacting with the system, but they do not need to be serviced as quickly as tasks in the user-interactive queue.

- **QOS_CLASS.Utility** —The **utility** class represents tasks that require a longer time to complete but do not demand immediate results. This class includes work such as importing data.
- **QOS_CLASS.Background** —Tasks belonging to the **background** class are not visible to the user and are not time sensitive. Examples include indexing a mailbox system and performing backups.

Tasks submitted to dispatch queues may be expressed in one of two different ways:

1. For the C, C++, and Objective-C languages, GCD identifies a language extension known as a **block**, which is simply a self-contained unit of work. A block is specified by a caret ^ inserted in front of a pair of braces { }. Code within the braces identifies the unit of work to be performed. A simple example of a block is shown below:

```
^{ printf("I am a block"); }
```

2. For the Swift programming language, a task is defined using a **closure**, which is similar to a block in that it expresses a self-contained unit of functionality. Syntactically, a Swift closure is written in the same way as a block, minus the leading caret.

The following Swift code segment illustrates obtaining a concurrent queue for the user-initiated class and submitting a task to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue
(QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue, { print("I am a closure.") })
```

Internally, GCD's thread pool is composed of POSIX threads. GCD actively manages the pool, allowing the number of threads to grow and shrink according to application demand and system capacity. GCD is implemented by the `libdispatch` library, which Apple has released under the Apache Commons license. It has since been ported to the FreeBSD operating system.

4.5.5 Intel Thread Building Blocks

Intel threading building blocks (TBB) is a template library that supports designing parallel applications in C++. As this is a library, it requires no special compiler or language support. Developers specify tasks that can run in par-

allel, and the TBB task scheduler maps these tasks onto underlying threads. Furthermore, the task scheduler provides load balancing and is cache aware, meaning that it will give precedence to tasks that likely have their data stored in cache memory and thus will execute more quickly. TBB provides a rich set of features, including templates for parallel loop structures, atomic operations, and mutual exclusion locking. In addition, it provides concurrent data structures, including a hash map, queue, and vector, which can serve as equivalent thread-safe versions of the C++ standard template library data structures.

Let's use parallel for loops as an example. Initially, assume there is a function named `apply(float value)` that performs an operation on the parameter `value`. If we had an array `v` of size `n` containing `float` values, we could use the following serial for loop to pass each value in `v` to the `apply()` function:

```
for (int i = 0; i < n; i++) {
    apply(v[i]);
}
```

A developer could manually apply data parallelism (Section 4.2.2) on a multicore system by assigning different regions of the array `v` to each processing core; however, this ties the technique for achieving parallelism closely to the physical hardware, and the algorithm would have to be modified and recompiled for the number of processing cores on each specific architecture.

Alternatively, a developer could use TBB, which provides a `parallel_for` template that expects two values:

```
parallel_for (range body)
```

where `range` refers to the range of elements that will be iterated (known as the [iteration space](#)) and `body` specifies an operation that will be performed on a subrange of elements.

We can now rewrite the above serial for loop using the TBB `parallel_for` template as follows:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

The first two parameters specify that the iteration space is from 0 to $n - 1$ (which corresponds to the number of elements in the array `v`). The second parameter is a C++ lambda function that requires a bit of explanation. The expression `[=](size_t i)` is the parameter `i`, which assumes each of the values over the iteration space (in this case from 0 to $n - 1$). Each value of `i` is used to identify which array element in `v` is to be passed as a parameter to the `apply(v[i])` function.

The TBB library will divide the loop iterations into separate “chunks” and create a number of tasks that operate on those chunks. (The `parallel_for` function allows developers to manually specify the size of the chunks if they wish to.) TBB will also create a number of threads and assign tasks to available threads. This is quite similar to the fork-join library in Java. The advantage of this approach is that it requires only that developers identify what operations can run in parallel (by specifying a `parallel_for` loop), and the library man-

ages the details involved in dividing the work into separate tasks that run in parallel. Intel TBB has both commercial and open-source versions that run on Windows, Linux, and macOS. Refer to the bibliography for further details on how to develop parallel applications using TBB.

4.6 Threading Issues

In this section, we discuss some of the issues to consider in designing multithreaded programs.

4.6.1 The fork() and exec() System Calls

In Chapter 3, we described how the `fork()` system call is used to create a separate, duplicate process. The semantics of the `fork()` and `exec()` system calls change in a multithreaded program.

If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call.

The `exec()` system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the `exec()` system call, the program specified in the parameter to `exec()` will replace the entire process—including all threads.

Which of the two versions of `fork()` to use depends on the application. If `exec()` is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to `exec()` will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec()` after forking, the separate process should duplicate all threads.

4.6.2 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as `<control><C>`) and

having a timer expire. Typically, an asynchronous signal is sent to another process.

A signal may be *handled* by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals may be ignored, while others (for example, an illegal memory access) are handled by terminating the program.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends on the type of signal generated. For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process. However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads.

The standard UNIX function for delivering a signal is

```
kill(pid_t pid, int signal)
```

This function specifies the process (`pid`) to which a particular signal (`signal`) is to be delivered. Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it. POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (`tid`):

```
pthread_kill(pthread_t tid, int signal)
```

Although Windows does not explicitly provide support for signals, it allows us to emulate them using **asynchronous procedure calls (APCs)**. The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated

by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process.

4.6.3 Thread Cancellation

Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation.** One thread immediately terminates the target thread.
2. **Deferred cancellation.** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.

In Pthreads, thread cancellation is initiated using the `pthread_cancel()` function. The identifier of the target thread is passed as a parameter to the function. The following code illustrates creating—and then canceling—a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Invoking `pthread_cancel()` indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request. When the target thread is finally canceled, the call to `pthread_join()` in the canceling thread returns. Pthreads supports three cancellation modes. Each mode is defined as a state and a type, as illustrated in the table below. A thread may set its cancellation state and type using an API.

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

As the table illustrates, Pthreads allows threads to disable or enable cancellation. Obviously, a thread cannot be canceled if cancellation is disabled. However, cancellation requests remain pending, so the thread can later enable cancellation and respond to the request.

The default cancellation type is deferred cancellation. However, cancellation occurs only when a thread reaches a **cancellation point**. Most of the blocking system calls in the POSIX and standard C library are defined as cancellation points, and these are listed when invoking the command `man pthreads` on a Linux system. For example, the `read()` system call is a cancellation point that allows cancelling a thread that is blocked while awaiting input from `read()`.

One technique for establishing a cancellation point is to invoke the `pthread_testcancel()` function. If a cancellation request is found to be pending, the call to `pthread_testcancel()` will not return, and the thread will terminate; otherwise, the call to the function will return, and the thread will continue to run. Additionally, Pthreads allows a function known as a **cleanup handler** to be invoked if a thread is canceled. This function allows any resources a thread may have acquired to be released before the thread is terminated.

The following code illustrates how a thread may respond to a cancellation request using deferred cancellation:

```

while (1) {
    /* do some work for awhile */

    . . .

    /* check if there is a cancellation request */
    pthread_testcancel();
}

```

Because of the issues described earlier, asynchronous cancellation is not recommended in Pthreads documentation. Thus, we do not cover it here. An interesting note is that on Linux systems, thread cancellation using the Pthreads API is handled through signals (Section 4.6.2).

Thread cancellation in Java uses a policy similar to deferred cancellation in Pthreads. To cancel a Java thread, you invoke the `interrupt()` method, which sets the interruption status of the target thread to true:

```

Thread worker;

. . .

/* set the interruption status of the thread */
worker.interrupt()

```

A thread can check its interruption status by invoking the `isInterrupted()` method, which returns a boolean value of a thread's interruption status:

```

while (!Thread.currentThread().isInterrupted()) {
    . . .
}

```

4.6.4 Thread-Local Storage

Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-local storage** (or **TLS**). For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique transaction identifier, we could use thread-local storage.

It is easy to confuse TLS with local variables. However, local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations. Additionally, when the developer has no control over the thread creation process—for example, when using an implicit technique such as a thread pool—then an alternative approach is necessary.

In some ways, TLS is similar to static data; the difference is that TLS data are unique to each thread. (In fact, TLS is usually declared as `static`.) Most thread libraries and compilers provide support for TLS. For example, Java provides a `ThreadLocal<T>` class with `set()` and `get()` methods for `ThreadLocal<T>` objects. Pthreads includes the type `pthread_key_t`, which provides a key that is specific to each thread. This key can then be used to access TLS data. Microsoft's C# language simply requires adding the `storage` attribute `[ThreadStatic]` to declare thread-local data. The `gcc` compiler provides the `storage` class keyword `_thread` for declaring TLS data. For example, if we wished to assign a unique identifier for each thread, we would declare it as follows:

```
static __thread int threadID;
```

4.6.5 Scheduler Activations

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required

by the many-to-many and two-level models discussed in Section 4.3.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a **lightweight process**, or **LWP**—is shown in Figure 4.20. To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only one thread can run at a time, so one LWP is sufficient. An application that is I/O-intensive may require multiple LWPs to execute, however. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.

One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the

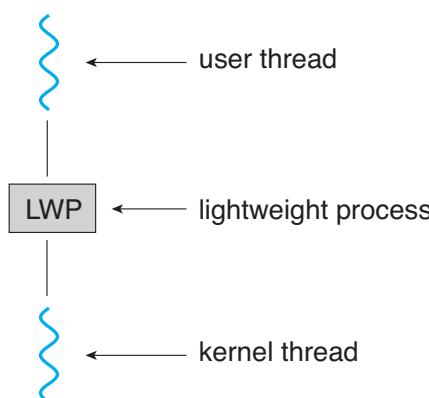


Figure 4.20 Lightweight process (LWP).

state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

4.7 Operating-System Examples

At this point, we have examined a number of concepts and issues related to threads. We conclude the chapter by exploring how threads are implemented in Windows and Linux systems.

4.7.1 Windows Threads

A Windows application runs as a separate process, and each process may contain one or more threads. The Windows API for creating threads is covered in Section 4.4.2. Additionally, Windows uses the one-to-one mapping described in Section 4.3.2, where each user-level thread maps to an associated kernel thread.

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor
- A program counter
- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
- A private storage area used by various run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the **context** of the thread.

The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB—thread environment block

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

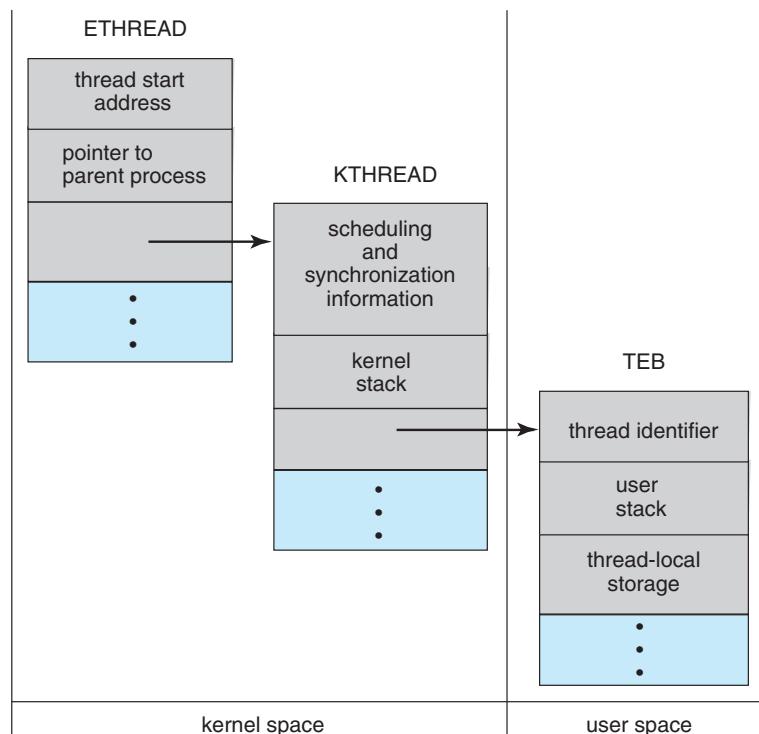


Figure 4.21 Data structures of a Windows thread.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains the thread identifier, a user-mode stack, and an array for thread-local storage. The structure of a Windows thread is illustrated in Figure 4.21.

4.7.2 Linux Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process, as described in Chapter 3. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program.

When `clone()` is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed in Figure 4.22. For example, suppose that `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`. The parent and child tasks will then share the same file-system information (such as the current working directory), the same memory space, the same signal handlers,

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figure 4.22 Some of the flags passed when `clone()` is invoked.

and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task. However, if none of these flags is set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to that provided by the `fork()` system call.

The varying level of sharing is possible because of the way a task is represented in the Linux kernel. A unique kernel data structure (specifically, `struct task_struct`) exists for each task in the system. This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored—for example, data structures that represent the list of open files, signal-handling information, and virtual memory. When `fork()` is invoked, a new task is created, along with a *copy* of all the associated data structures of the parent process. A new task is also created when the `clone()` system call is made. However, rather than copying all data structures, the new task *points* to the data structures of the parent task, depending on the set of flags passed to `clone()`.

Finally, the flexibility of the `clone()` system call can be extended to the concept of containers, a virtualization topic which was introduced in Chapter 1. Recall from that chapter that a container is a virtualization technique provided by the operating system that allows creating multiple Linux systems (containers) under a single Linux kernel that run in isolation to one another. Just as certain flags passed to `clone()` can distinguish between creating a task that behaves more like a process or a thread based upon the amount of sharing between the parent and child tasks, there are other flags that can be passed to `clone()` that allow a Linux container to be created. Containers will be covered more fully in Chapter 18.

4.8 Summary

- A thread represents a basic unit of CPU utilization, and threads belonging to the same process share many of the process resources, including code and data.
- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- Concurrency exists when multiple threads are making progress, whereas parallelism exists when multiple threads are making progress simulta-

neously. On a system with a single CPU, only concurrency is possible; parallelism requires a multicore system that provides multiple CPUs.

- There are several challenges in designing multithreaded applications. They include dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies. Finally, multithreaded programs are especially challenging to test and debug.
- Data parallelism distributes subsets of the same data across different computing cores and performs the same operation on each core. Task parallelism distributes not data but tasks across multiple cores. Each task is running a unique operation.
- User applications create user-level threads, which must ultimately be mapped to kernel threads to execute on a CPU. The many-to-one model maps many user-level threads to one kernel thread. Other approaches include the one-to-one and many-to-many models.
- A thread library provides an API for creating and managing threads. Three common thread libraries include Windows, Pthreads, and Java threading. Windows is for the Windows system only, while Pthreads is available for POSIX-compatible systems such as UNIX, Linux, and macOS. Java threads will run on any system that supports a Java virtual machine.
- Implicit threading involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads. There are several approaches to implicit threading, including thread pools, fork-join frameworks, and Grand Central Dispatch. Implicit threading is becoming an increasingly common technique for programmers to use in developing concurrent and parallel applications.
- Threads may be terminated using either asynchronous or deferred cancellation. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.
- Unlike many other operating systems, Linux does not distinguish between processes and threads; instead, it refers to each as a task. The Linux `clone()` system call can be used to create tasks that behave either more like processes or more like threads.

Practice Exercises

- 4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.
- 4.2 Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

- 4.3 Does the multithreaded web server described in Section 4.1 exhibit task or data parallelism?
- 4.4 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?
- 4.5 Describe the actions taken by a kernel to context-switch between kernel-level threads.
- 4.6 What resources are used when a thread is created? How do they differ from those used when a process is created?
- 4.7 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

Further Reading

[Vahalia (1996)] covers threading in several versions of UNIX. [McDougall and Mauro (2007)] describes developments in threading the Solaris kernel. [Russinovich et al. (2017)] discuss threading in the Windows operating system family. [Mauerer (2008)] and [Love (2010)] explain how Linux handles threading, and [Levin (2013)] covers threads in macOS and iOS. [Herlihy and Shavit (2012)] covers parallelism issues on multicore systems. [Aubanel (2017)] covers parallelism of several different algorithms.

Bibliography

- [Aubanel (2017)] E. Aubanel, *Elements of Parallel Computing*, CRC Press (2017).
- [Herlihy and Shavit (2012)] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).
- [Levin (2013)] J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).
- [Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).
- [Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).
- [McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).
- [Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).
- [Vahalia (1996)] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall (1996).

Chapter 4 Exercises

- 4.8 Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution.
- 4.9 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?
- 4.10 Which of the following components of program state are shared across threads in a multithreaded process?
 - a. Register values
 - b. Heap memory
 - c. Global variables
 - d. Stack memory
- 4.11 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.
- 4.12 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.
- 4.13 Is it possible to have concurrency but not parallelism? Explain.
- 4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:
 - 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores
 - 67 percent parallel with (a) two processing cores and (b) four processing cores
 - 90 percent parallel with (a) four processing cores and (b) eight processing cores
- 4.15 Determine if the following problems exhibit task or data parallelism:
 - Using a separate thread to generate a thumbnail for each photo in a collection
 - Transposing a matrix in parallel
 - A networked application where one thread reads from the network and another writes to the network
 - The fork-join array summation application described in Section 4.5.2
 - The Grand Central Dispatch system
- 4.16 A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be

opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

4.17 Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
    fork();
    thread_create( . . . );
}
fork();
```

- a. How many unique processes are created?
 - b. How many unique threads are created?
- 4.18** As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. However, other operating systems, such as Windows, treat processes and threads differently. Typically, such systems use a notation in which the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.
- 4.19** The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?
- 4.20** Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

- a. The number of kernel threads allocated to the program is less than the number of processing cores.
- b. The number of kernel threads allocated to the program is equal to the number of processing cores.
- c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

```

#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork();

    if (pid == 0) { /* child process */
        pthread_attr_init(&attr);
        pthread_create(&tid,&attr,runner,NULL);
        pthread_join(tid,NULL);
        printf("CHILD: value = %d",value); /* LINE C */
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE P */
    }
}

void *runner(void *param) {
    value = 5;
    pthread_exit(0);
}

```

Figure 4.22 C program for Exercise 4.19.

- 4.21** Pthreads provides an API for managing thread cancellation. The `pthread_setcancelstate()` function is used to set the cancellation state. Its prototype appears as follows:

```
pthread_setcancelstate(int state, int *oldstate)
```

The two possible values for the state are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

Using the code segment shown in Figure 4.24, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

```
int oldstate;  
  
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);  
  
/* What operations would be performed here? */  
  
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Figure 4.23 C program for Exercise 4.21.

Programming Problems

- 4.22 Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

```
The average value is 82  
The minimum value is 72  
The maximum value is 95
```

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

- 4.23 Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

- 4.24 An interesting way of calculating π is to use a technique known as *Monte Carlo*, which involves randomization. This technique works as follows:

Suppose you have a circle inscribed within a square, as shown in Figure 4.25. (Assume that the radius of this circle is 1.)

- First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle.
- Next, estimate π by performing the following calculation:

$$\pi = 4 \times (\text{number of points in circle}) / (\text{total number of points})$$

Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store that result in a global variable. When this thread has exited, the parent thread will calculate and output the estimated value of π . It is worth experimenting with the number of random points generated. As a general rule, the greater the number of points, the closer the approximation to π .

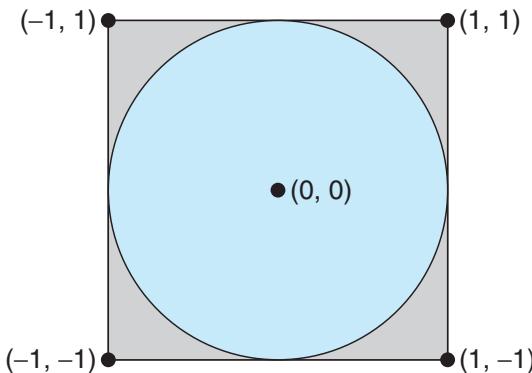


Figure 4.25 Monte Carlo technique for calculating π .

In the source-code download for this text, you will find a sample program that provides a technique for generating random numbers, as well as determining if the random (x, y) point occurs within the circle.

Readers interested in the details of the Monte Carlo method for estimating π should consult the bibliography at the end of this chapter. In Chapter 6, we modify this exercise using relevant material from that chapter.

- 4.25 Repeat Exercise 4.24, but instead of using a separate thread to generate random points, use OpenMP to parallelize the generation of points. Be careful not to place the calculation of π in the parallel region, since you want to calculate π only once.
- 4.26 Modify the socket-based date server (Figure 3.27) in Chapter 3 so that the server services each client request in a separate thread.
- 4.27 The Fibonacci sequence is the series of numbers $0, 1, 1, 2, 3, 5, 8, \dots$. Formally, it can be expressed as:

$$\begin{aligned}fib_0 &= 0 \\fib_1 &= 1 \\fib_n &= fib_{n-1} + fib_{n-2}\end{aligned}$$

Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish. Use the techniques described in Section 4.4 to meet this requirement.

- 4.28 Modify programming problem Exercise 3.20 from Chapter 3, which asks you to design a pid manager. This modification will consist of writing a

multithreaded program that tests your solution to Exercise 3.20. You will create a number of threads—for example, 100—and each thread will request a pid, sleep for a random period of time, and then release the pid. (Sleeping for a random period of time approximates the typical pid usage in which a pid is assigned to a new process, the process executes and then terminates, and the pid is released on the process's termination.) On UNIX and Linux systems, sleeping is accomplished through the `sleep()` function, which is passed an integer value representing the number of seconds to sleep. This problem will be modified in Chapter 7.

- 4.29** Exercise 3.25 in Chapter 3 involves designing an echo server using the Java threading API. This server is single-threaded, meaning that the server cannot respond to concurrent echo clients until the current client exits. Modify the solution to Exercise 3.25 so that the echo server services each client in a separate request.

Programming Projects

Project 1—Sudoku Solution Validator

A *Sudoku* puzzle uses a 9×9 grid in which each column and row, as well as each of the nine 3×3 subgrids, must contain all of the digits 1 · · · 9. Figure 4.26 presents an example of a valid Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid.

There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 4.26 Solution to a 9×9 Sudoku puzzle.

- Nine threads to check that each of the 3×3 subgrids contains the digits 1 through 9

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this project. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

I. Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a `struct`. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
/* structure for passing data to threads */
typedef struct
{
    int row;
    int column;
} parameters;
```

Both Pthreads and Windows programs will create worker threads using a strategy similar to that shown below:

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Now create the thread passing it data as a parameter */
```

The data pointer will be passed to either the `pthread_create()` (Pthreads) function or the `CreateThread()` (Windows) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

II. Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The i^{th} index in this array corresponds to the i^{th} worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 indicates otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

Project 2—Multithreaded Sorting Application

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we

will term *sorting threads*) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a *merging thread*—which merges the two sublists into a single sorted list.

Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as in Figure 4.27.

This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions in Project 1 for details on passing parameters to a thread.

The parent thread will output the sorted array once all sorting threads have exited.

Project 3—Fork-Join Sorting Application

Implement the preceding project (Multithreaded Sorting Application) using Java’s fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting algorithm:

1. Quicksort
2. Mergesort

The Quicksort implementation will use the Quicksort algorithm for dividing the list of elements to be sorted into a *left half* and a *right half* based on the

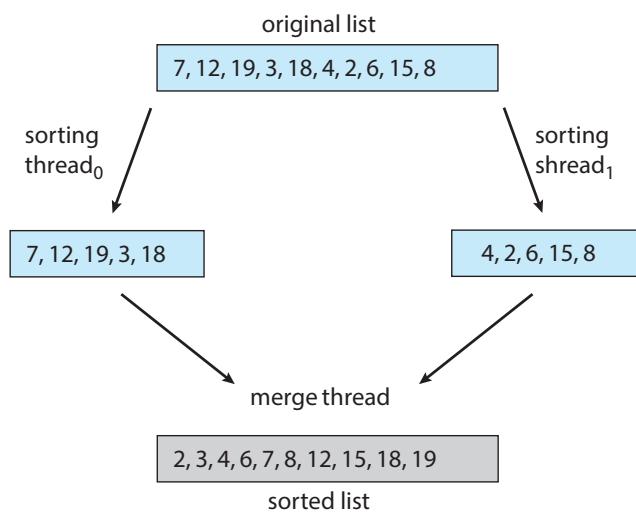


Figure 4.27 Multithreaded sorting.

position of the pivot value. The Mergesort algorithm will divide the list into two evenly sized halves. For both the Quicksort and Mergesort algorithms, when the list to be sorted falls within some threshold value (for example, the list is size 100 or fewer), directly apply a simple algorithm such as the Selection or Insertion sort. Most data structures texts describe these two well-known, divide-and-conquer sorting algorithms.

The class `SumTask` shown in Section 4.5.2.1 extends `RecursiveTask`, which is a result-bearing `ForkJoinTask`. As this assignment will involve sorting the array that is passed to the task, but not returning any values, you will instead create a class that extends `RecursiveAction`, a non result-bearing `ForkJoinTask` (see Figure 4.19).

The objects passed to each sorting algorithm are required to implement Java's `Comparable` interface, and this will need to be reflected in the class definition for each sorting algorithm. The source code download for this text includes Java code that provides the foundations for beginning this project.

CPU Scheduling



CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms, including real-time systems. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 4, we introduced threads to the process model. On modern operating systems it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms "process scheduling" and "thread scheduling" are often used interchangeably. In this chapter, we use *process scheduling* when discussing general scheduling concepts and *thread scheduling* to refer to thread-specific ideas.

Similarly, in Chapter 1 we describe how a *core* is the basic computational unit of a CPU, and that a process executes on a CPU's core. However, in many instances in this chapter, when we use the general terminology of scheduling a process to "run on a CPU", we are implying that the process is running on a CPU's core.

CHAPTER OBJECTIVES

- Describe various CPU scheduling algorithms.
- Assess CPU scheduling algorithms based on scheduling criteria.
- Explain the issues related to multiprocessor and multicore scheduling.
- Describe various real-time scheduling algorithms.
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems.
- Apply modeling and simulations to evaluate CPU scheduling algorithms.
- Design a program that implements several different CPU scheduling algorithms.

5.1 Basic Concepts

In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU's core is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

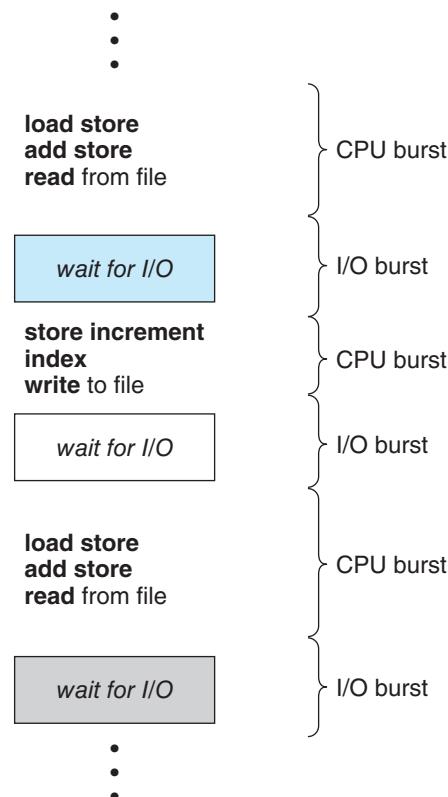


Figure 5.1 Alternating sequence of CPU and I/O bursts.

5.1.1 CPU-I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 5.2. The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.

5.1.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **CPU scheduler**, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

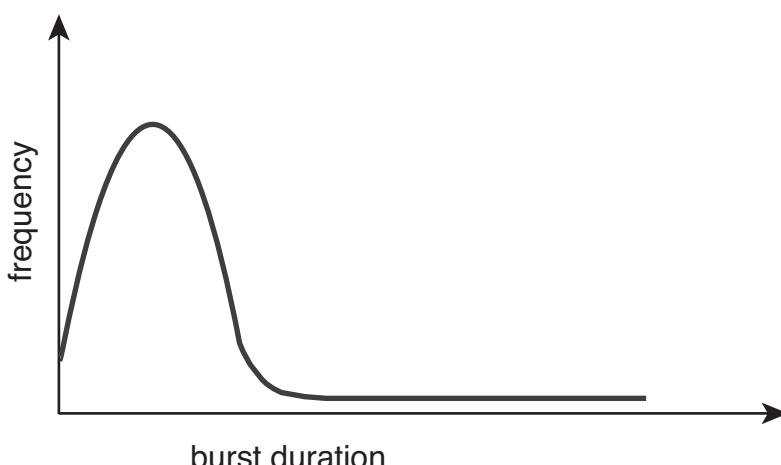


Figure 5.2 Histogram of CPU-burst durations.

5.1.3 Preemptive and Nonpreemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of `wait()` for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. This issue will be explored in detail in Chapter 6.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. As will be discussed in Section 6.2, operating-system kernels can be designed as either nonpreemptive or preemptive. A nonpreemptive kernel will wait for a system call to complete or for a process to block while waiting for I/O to complete to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing, where tasks must complete execution within a given time frame. In Section 5.6, we explore scheduling demands of real-time systems. A preemptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structures. Most modern operating systems are now fully preemptive when running in kernel mode.

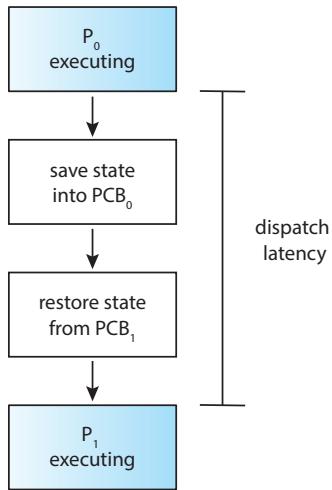


Figure 5.3 The role of the dispatcher.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times. Otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

5.1.4 Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. This function involves the following:

- Switching context from one process to another
 - Switching to user mode
 - Jumping to the proper location in the user program to resume that program

The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency** and is illustrated in Figure 5.3.

An interesting question to consider is, how often do context switches occur? On a system-wide level, the number of context switches can be obtained by using the `vmstat` command that is available on Linux systems. Below is the output (which has been trimmed) from the command

```
vmstat 1 3
```

This command provides 3 lines of output over a 1-second delay:

```
-----cpu-----
24
225
339
```

The first line gives the average number of context switches over 1 second since the system booted, and the next two lines give the number of context switches over two 1-second intervals. Since this machine booted, it has averaged 24 context switches per second. And in the past second, 225 context switches were made, with 339 context switches in the second prior to that.

We can also use the /proc file system to determine the number of context switches for a given process. For example, the contents of the file /proc/2166/status will list various statistics for the process with pid = 2166. The command

```
cat /proc/2166/status
```

provides the following trimmed output:

voluntary_ctxt_switches	150
nonvoluntary_ctxt_switches	8

This output shows the number of context switches over the lifetime of the process. Notice the distinction between *voluntary* and *nonvoluntary* context switches. A voluntary context switch occurs when a process has given up control of the CPU because it requires a resource that is currently unavailable (such as blocking for I/O.) A nonvoluntary context switch occurs when the CPU has been taken away from a process, such as when its time slice has expired or it has been preempted by a higher-priority process.

5.2 Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization.** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). (CPU utilization can be obtained by using the top command on Linux, macOS, and UNIX systems.)
- **Throughput.** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed

per time unit, called **throughput**. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

- **Turnaround time.** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time.** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time.** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as a PC desktop or laptop system), it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

As we discuss various CPU-scheduling algorithms in the following section, we illustrate their operation. An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time. More elaborate evaluation mechanisms are discussed in Section 5.8.

5.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPU-scheduling algorithms. In this section, we describe several of them. Although most modern CPU architectures have multiple processing cores, we describe these scheduling algorithms in the context of only one processing core available. That is, a single CPU that has a single processing core, thus the system is

capable of only running one process at a time. In Section 5.5 we discuss CPU scheduling in the context of multiprocessor systems.

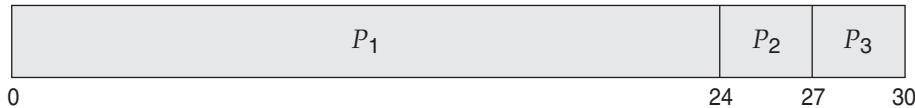
5.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the [first-come first-served \(FCFS\)](#) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

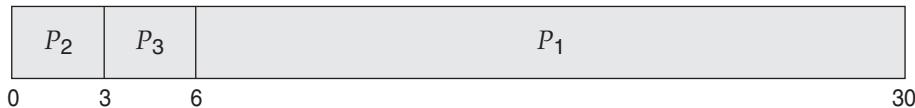
On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3 , and are served in FCFS order, we get the result shown in the following [Gantt chart](#), which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P_1 , 24 milliseconds for process P_2 , and 27 milliseconds for process P_3 . Thus, the average waiting time is $(0 + 24 + 27)/3 = 17$ milliseconds. If the processes arrive in the order P_2, P_3, P_1 , however, the results will be as shown in the following Gantt chart:



The average waiting time is now $(6 + 0 + 3)/3 = 3$ milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O

devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

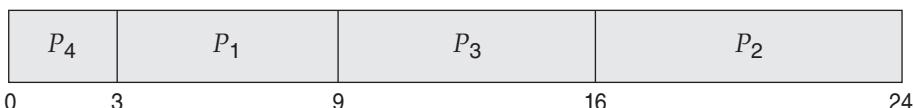
5.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-first (SJF)** scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the *shortest-next-CPU-burst* algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P_1 , 16 milliseconds for process P_2 , 9 milliseconds for process P_3 , and 0 milliseconds for process P_4 . Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process

before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let t_n be the length of the n th CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α , $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

The value of t_n contains our most recent information, while τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ_0 can be defined as a constant or as an overall system average. Figure 5.4 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.

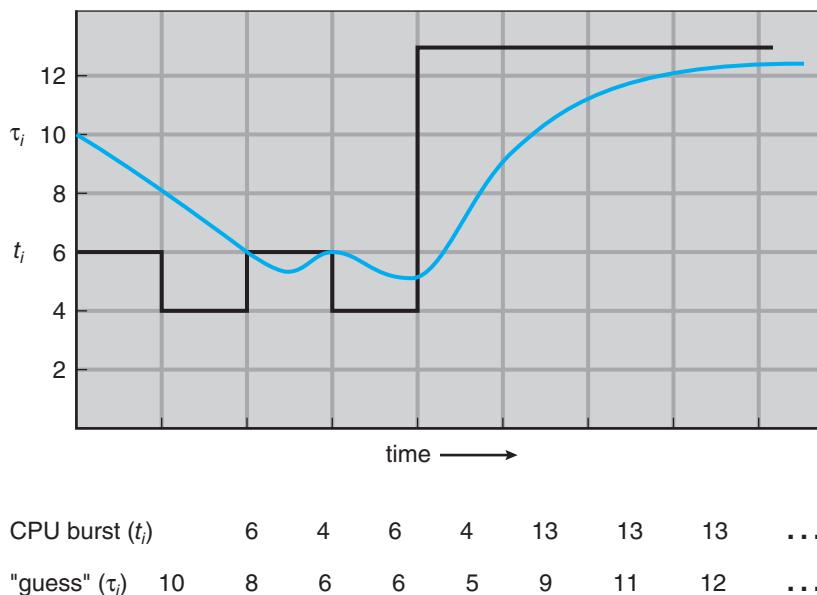


Figure 5.4 Prediction of the length of the next CPU burst.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n to find

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

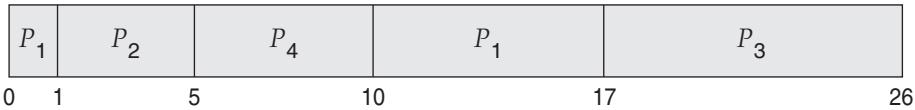
Typically, α is less than 1. As a result, $(1 - \alpha)$ is also less than 1, and each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

5.3.3 Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.

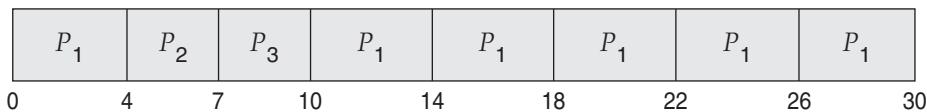
The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2 . Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3 . Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds ($10 - 4$), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is $17/3 = 5.66$ milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large

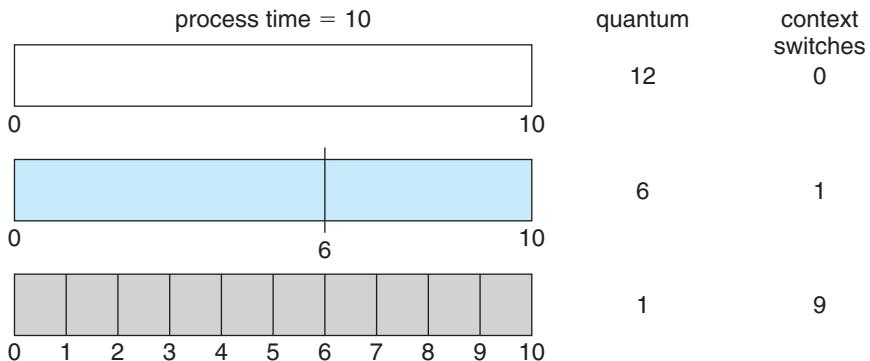


Figure 5.5 How a smaller time quantum increases context switches.

number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.5).

Thus, we want the time quantum to be large with respect to the context-switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.6, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context-switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

5.3.4 Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the

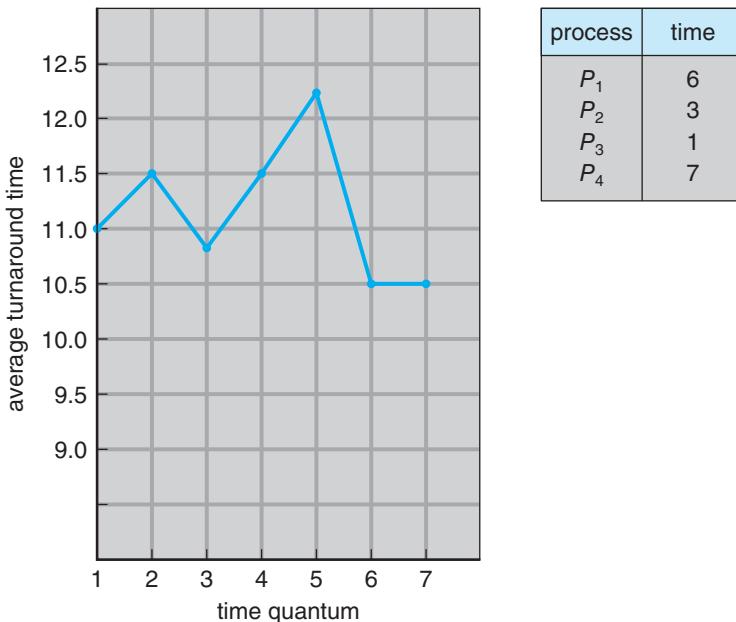


Figure 5.6 How turnaround time varies with the time quantum.

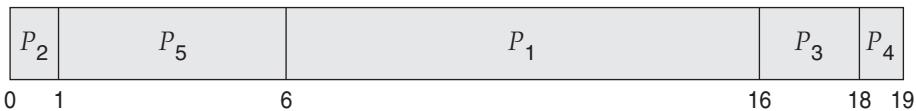
process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of *high* priority and *low* priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P_1, P_2, \dots, P_5 , with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

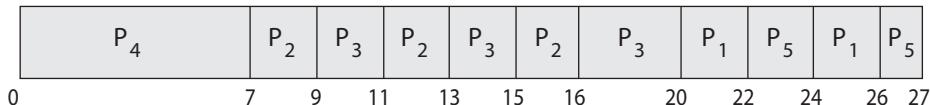
A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take a little over 2 minutes for a priority-127 process to age to a priority-0 process.

Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling. Let's illustrate with an example using the following set of processes, with the burst time in milliseconds:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:



In this example, process P_4 has the highest priority, so it will run to completion. Processes P_2 and P_3 have the next-highest priority, and they will execute in a round-robin fashion. Notice that when process P_2 finishes at time 16, process P_3 is the highest-priority process, so it will run until it completes execution. Now, only processes P_1 and P_5 remain, and as they have equal priority, they will execute in round-robin order until they complete.

5.3.5 Multilevel Queue Scheduling

With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run. Depending on how the queues are managed, an $O(n)$ search may be necessary to determine the highest-priority process. In practice, it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue. This is illustrated in Figure 5.7. This approach—known as **multilevel queue**—also works well when priority scheduling is combined with round-robin: if there are multiple processes in the highest-priority queue, they are executed in round-robin order. In the most generalized form of this approach, a priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime.

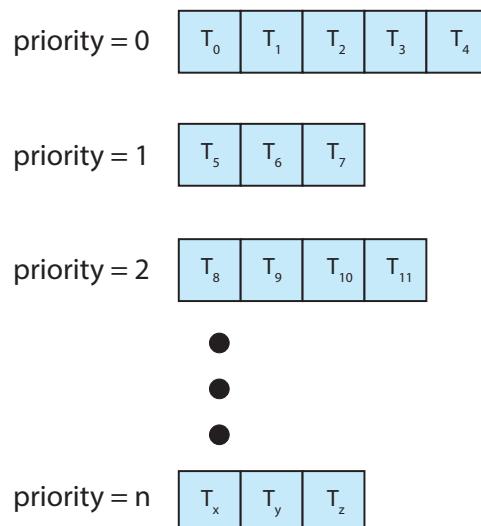


Figure 5.7 Separate queues for each priority.

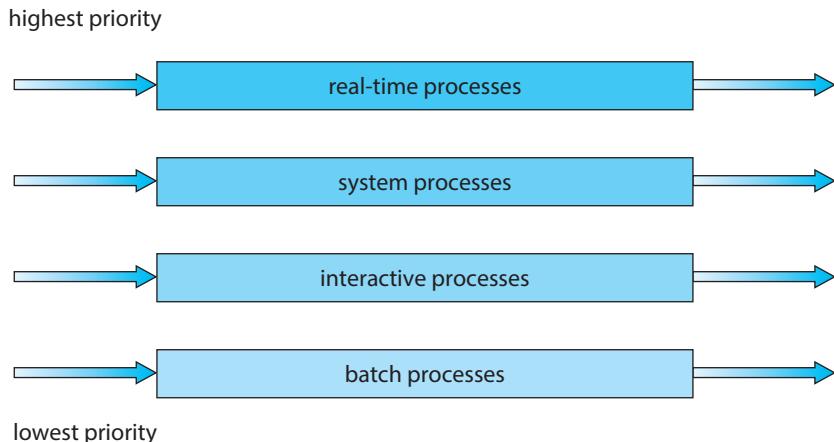


Figure 5.8 Multilevel queue scheduling.

A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type (Figure 5.8). For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes. Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm. The foreground queue might be scheduled by an RR algorithm, for example, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the real-time queue may have absolute priority over the interactive queue.

Let's look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

1. Real-time processes
2. System processes
3. Interactive processes
4. Batch processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling

among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

5.3.6 Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts—in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.9). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

An entering process is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty. To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

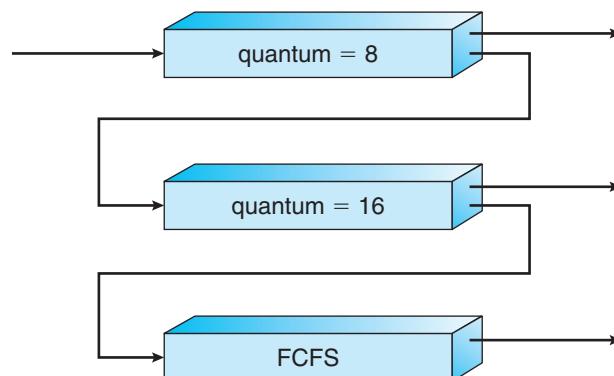


Figure 5.9 Multilevel feedback queues.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

5.4 Thread Scheduling

In Chapter 4, we introduced threads to the process model, distinguishing between *user-level* and *kernel-level* threads. On most modern operating systems it is kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

5.4.1 Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one (Section 4.3.1) and many-to-many (Section 4.3.3) models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process-contention scope (PCS)**, since competition for the CPU takes place among threads belonging to the same process. (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually *running* on a CPU as that further requires the operating system to schedule the LWP’s kernel thread onto a physical CPU core.) To decide which

kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**. Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model (Section 4.3.2), such as Windows and Linux schedule threads using only SCS.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing (Section 5.3.3) among threads of equal priority.

5.4.2 Pthread Scheduling

We provided a sample POSIX Pthread program in Section 4.4.1, along with an introduction to thread creation with Pthreads. Now, we highlight the POSIX Pthread API that allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations (Section 4.6.5). The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC (Interprocess Communication) provides two functions for setting—and getting—the contention scope policy:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the `pthread_attr_setscope()` function is passed either the PTHREAD_SCOPE_SYSTEM or the PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of `pthread_attr_getscope()`, this second parameter contains a pointer to an `int` value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a nonzero value.

In Figure 5.10, we illustrate a Pthread scheduling API. The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM. It then creates five separate threads that will run using the SCS scheduling policy. Note that on some systems, only certain contention scope values are allowed. For example, Linux and macOS systems allow only PTHREAD_SCOPE_SYSTEM.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }

    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */

    pthread_exit(0);
}
```

Figure 5.10 Pthread scheduling API.

5.5 Multi-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processing core. If multiple CPUs are available, **load sharing**, where multiple threads may run in parallel, becomes possible, however scheduling issues become correspondingly more complex. Many possibilities have been tried; and as we saw with CPU scheduling with a single-core CPU, there is no one best solution.

Traditionally, the term **multiprocessor** referred to systems that provided multiple physical processors, where each processor contained one single-core CPU. However, the definition of multiprocessor has evolved significantly, and on modern computing systems, **multiprocessor** now applies to the following system architectures:

- Multicore CPUs
- Multithreaded cores
- NUMA systems
- Heterogeneous multiprocessing

Here, we discuss several concerns in multiprocessor scheduling in the context of these different architectures. In the first three examples we concentrate on systems in which the processors are identical—homogeneous—in terms of their functionality. We can then use any available CPU to run any process in the queue. In the last example we explore a system where the processors are not identical in their capabilities.

5.5.1 Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one core accesses the system data structures, reducing the need for data sharing. The downfall of this approach is the master server becomes a potential bottleneck where overall system performance may be reduced.

The standard approach for supporting multiprocessors is **symmetric multiprocessing (SMP)**, where each processor is self-scheduling. Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run. Note that this provides two possible strategies for organizing the threads eligible to be scheduled:

1. All threads may be in a common ready queue.
2. Each processor may have its own private queue of threads.

These two strategies are contrasted in Figure 5.11. If we select the first option, we have a possible race condition on the shared ready queue and therefore must ensure that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue. As discussed in

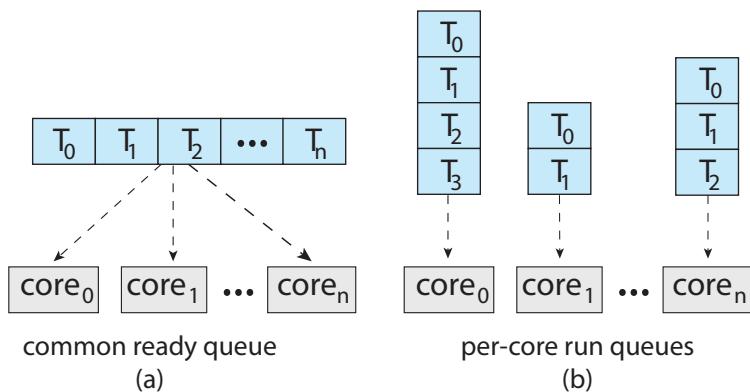


Figure 5.11 Organization of ready queues.

Chapter 6, we could use some form of locking to protect the common ready queue from this race condition. Locking would be highly contended, however, as all accesses to the queue would require lock ownership, and accessing the shared queue would likely be a performance bottleneck. The second option permits each processor to schedule threads from its private run queue and therefore does not suffer from the possible performance problems associated with a shared run queue. Thus, it is the most common approach on systems supporting SMP. Additionally, as described in Section 5.5.4, having private, per-processor run queues in fact may lead to more efficient use of cache memory. There are issues with per-processor run queues—most notably, workloads of varying sizes. However, as we shall see, balancing algorithms can be used to equalize workloads among all processors.

Virtually all modern operating systems support SMP, including Windows, Linux, and macOS as well as mobile systems including Android and iOS. In the remainder of this section, we discuss issues concerning SMP systems when designing CPU scheduling algorithms.

5.5.2 Multicore Processors

Traditionally, SMP systems have allowed several processes to run in parallel by providing multiple physical processors. However, most contemporary computer hardware now places multiple computing cores on the same physical chip, resulting in a **multicore processor**. Each core maintains its architectural state and thus appears to the operating system to be a separate logical CPU. SMP systems that use multicore processors are faster and consume less power than systems in which each CPU has its own physical chip.

Multicore processors may complicate scheduling issues. Let's consider how this can happen. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall**, occurs primarily because modern processors operate at much faster speeds than memory. However, a memory stall can also occur because of a cache miss (accessing data that are not in cache memory). Figure 5.12 illustrates a memory stall. In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory.

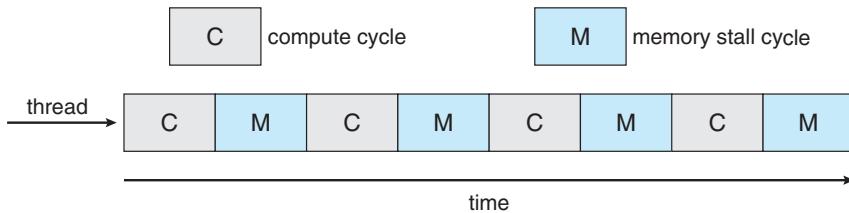


Figure 5.12 Memory stall.

To remedy this situation, many recent hardware designs have implemented multithreaded processing cores in which two (or more) **hardware threads** are assigned to each core. That way, if one hardware thread stalls while waiting for memory, the core can switch to another thread. Figure 5.13 illustrates a dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved. From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread. This technique—known as **chip multithreading** (CMT)—is illustrated in Figure 5.14. Here, the processor contains four computing cores, with each core containing two hardware threads. From the perspective of the operating system, there are eight logical CPUs.

Intel processors use the term **hyper-threading** (also known as **simultaneous multithreading** or SMT) to describe assigning multiple hardware threads to a single processing core. Contemporary Intel processors—such as the i7—support two threads per core, while the Oracle Sparc M7 processor supports eight threads per core, with eight cores per processor, thus providing the operating system with 64 logical CPUs.

In general, there are two ways to multithread a processing core: **coarse-grained** and **fine-grained** multithreading. With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions. Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity—typically at the boundary of an instruction

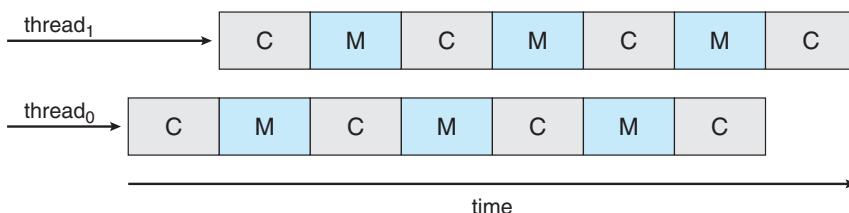


Figure 5.13 Multithreaded multicore system.

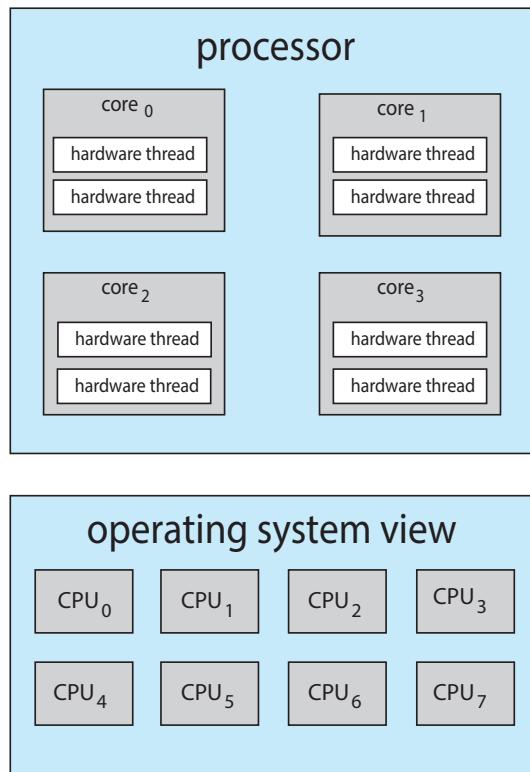


Figure 5.14 Chip multithreading.

cycle. However, the architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.

It is important to note that the resources of the physical core (such as caches and pipelines) must be shared among its hardware threads, and therefore a processing core can only execute one hardware thread at a time. Consequently, a multithreaded, multicore processor actually requires two different levels of scheduling, as shown in Figure 5.15, which illustrates a dual-threaded processing core.

On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical CPU). For all practical purposes, such decisions have been the primary focus of this chapter. Therefore, for this level of scheduling, the operating system may choose any scheduling algorithm, including those described in Section 5.3.

A second level of scheduling specifies how each core decides which hardware thread to run. There are several strategies to adopt in this situation. One approach is to use a simple round-robin algorithm to schedule a hardware thread to the processing core. This is the approach adopted by the UltraSPARC T3. Another approach is used by the Intel Itanium, a dual-core processor with two hardware-managed threads per core. Assigned to each hardware thread is a dynamic *urgency* value ranging from 0 to 7, with 0 representing the lowest urgency and 7 the highest. The Itanium identifies five different events that may

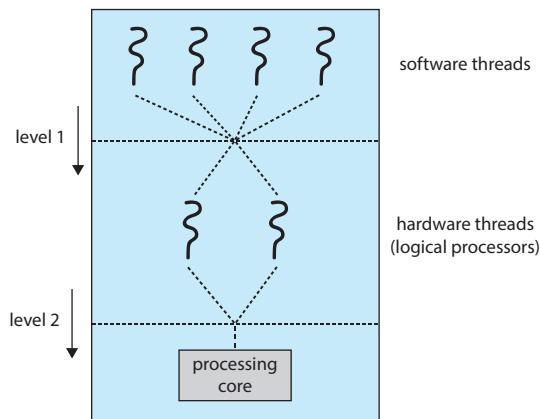


Figure 5.15 Two levels of scheduling.

trigger a thread switch. When one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core.

Note that the two different levels of scheduling shown in Figure 5.15 are not necessarily mutually exclusive. In fact, if the operating system scheduler (the first level) is made aware of the sharing of processor resources, it can make more effective scheduling decisions. As an example, assume that a CPU has two processing cores, and each core has two hardware threads. If two software threads are running on this system, they can be running either on the same core or on separate cores. If they are both scheduled to run on the same core, they have to share processor resources and thus are likely to proceed more slowly than if they were scheduled on separate cores. If the operating system is aware of the level of processor resource sharing, it can schedule software threads onto logical processors that do not share resources.

5.5.3 Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads, along with ready queues of threads awaiting the CPU. **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private ready queue of eligible threads to execute. On systems with a common run queue, load balancing is unnecessary, because once a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.

There are two general approaches to load balancing: push migration and pull migration. With **push migration**, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors. **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are, in fact, often implemented in parallel on load-balancing systems. For

example, the Linux CFS scheduler (described in Section 5.7.1) and the ULE scheduler available for FreeBSD systems implement both techniques.

The concept of a “balanced load” may have different meanings. One view of a balanced load may require simply that all queues have approximately the same number of threads. Alternatively, balance may require an equal distribution of thread priorities across all queues. In addition, in certain situations, neither of these strategies may be sufficient. Indeed, they may work against the goals of the scheduling algorithm. (We leave further consideration of this as an exercise.)

5.5.4 Processor Affinity

Consider what happens to cache memory when a thread has been running on a specific processor. The data most recently accessed by the thread populate the cache for the processor. As a result, successive memory accesses by the thread are often satisfied in cache memory (known as a “warm cache”). Now consider what happens if the thread migrates to another processor—say, due to load balancing. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most operating systems with SMP support try to avoid migrating a thread from one processor to another and instead attempt to keep a thread running on the same processor and take advantage of a warm cache. This is known as **processor affinit** —that is, a process has an affinity for the processor on which it is currently running.

The two strategies described in Section 5.5.1 for organizing the queue of threads available for scheduling have implications for processor affinity. If we adopt the approach of a common ready queue, a thread may be selected for execution by any processor. Thus, if a thread is scheduled on a new processor, that processor’s cache must be repopulated. With private, per-processor ready queues, a thread is always scheduled on the same processor and can therefore benefit from the contents of a warm cache. Essentially, per-processor ready queues provide processor affinity for free!

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as **soft affinit** . Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing. In contrast, some systems provide system calls that support **hard affinit** , thereby allowing a process to specify a subset of processors on which it can run. Many systems provide both soft and hard affinity. For example, Linux implements soft affinity, but it also provides the `sched_setaffinity()` system call, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run.

The main-memory architecture of a system can affect processor affinity issues as well. Figure 5.16 illustrates an architecture featuring non-uniform memory access (NUMA) where there are two physical processor chips each with their own CPU and local memory. Although a system interconnect allows all CPUs in a NUMA system to share one physical address space, a CPU has faster access to its local memory than to memory local to another CPU. If the operating system’s CPU scheduler and memory-placement algorithms are *NUMA-aware*

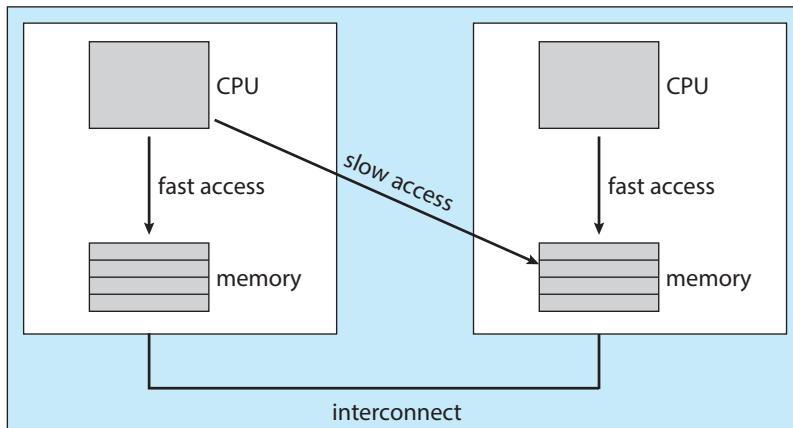


Figure 5.16 NUMA and CPU scheduling.

and work together, then a thread that has been scheduled onto a particular CPU can be allocated memory closest to where the CPU resides, thus providing the thread the fastest possible memory access.

Interestingly, load balancing often counteracts the benefits of processor affinity. That is, the benefit of keeping a thread running on the same processor is that the thread can take advantage of its data being in that processor's cache memory. Balancing loads by moving a thread from one processor to another removes this benefit. Similarly, migrating a thread between processors may incur a penalty on NUMA systems, where a thread may be moved to a processor that requires longer memory access times. In other words, there is a natural tension between load balancing and minimizing memory access times. Thus, scheduling algorithms for modern multicore NUMA systems have become quite complex. In Section 5.7.1, we examine the Linux CFS scheduling algorithm and explore how it balances these competing goals.

5.5.5 Heterogeneous Multiprocessing

In the examples we have discussed so far, all processors are identical in terms of their capabilities, thus allowing any thread to run on any processing core. The only difference being that memory access times may vary based upon load balancing and processor affinity policies, as well as on NUMA systems.

Although mobile systems now include multicore architectures, some systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core. Such systems are known as **heterogeneous multiprocessing** (HMP). Note this is not a form of asymmetric multiprocessing as described in Section 5.5.1 as both system and user tasks can run on any core. Rather, the intention behind HMP is to better manage power consumption by assigning tasks to certain cores based upon the specific demands of the task.

For ARM processors that support it, this type of architecture is known as **big.LITTLE** where higher-performance *big* cores are combined with energy efficient *LITTLE* cores. *Big* cores consume greater energy and therefore should

only be used for short periods of time. Likewise, *little* cores use less energy and can therefore be used for longer periods.

There are several advantages to this approach. By combining a number of slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but may need to run for longer periods, (such as background tasks) to little cores, thereby helping to preserve a battery charge. Similarly, interactive applications which require more processing power, but may run for shorter durations, can be assigned to big cores. Additionally, if the mobile device is in a power-saving mode, energy-intensive big cores can be disabled and the system can rely solely on energy-efficient little cores. Windows 10 supports HMP scheduling by allowing a thread to select a scheduling policy that best supports its power management demands.

5.6 Real-Time CPU Scheduling

CPU scheduling for real-time operating systems involves special issues. In general, we can distinguish between soft real-time systems and hard real-time systems. **Soft real-time systems** provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all. In this section, we explore several issues related to process scheduling in both soft and hard real-time operating systems.

5.6.1 Minimizing Latency

Consider the event-driven nature of a real-time system. The system is typically waiting for an event in real time to occur. Events may arise either in software—as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. We refer to **event latency** as the amount of time that elapses from when an event occurs to when it is serviced (Figure 5.17).

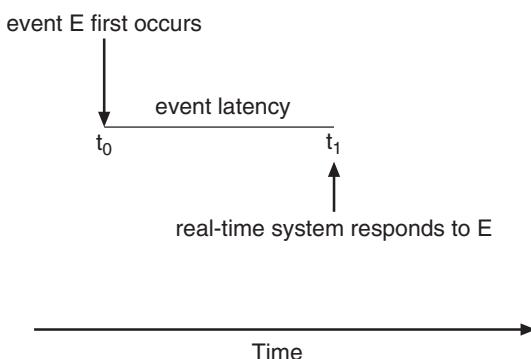


Figure 5.17 Event latency.

Usually, different events have different latency requirements. For example, the latency requirement for an antilock brake system might be 3 to 5 milliseconds. That is, from the time a wheel first detects that it is sliding, the system controlling the antilock brakes has 3 to 5 milliseconds to respond to and control the situation. Any response that takes longer might result in the automobile's veering out of control. In contrast, an embedded system controlling radar in an airliner might tolerate a latency period of several seconds.

Two types of latencies affect the performance of real-time systems:

1. Interrupt latency
2. Dispatch latency

Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure 5.18).

Obviously, it is crucial for real-time operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention. Indeed, for hard real-time systems, interrupt latency must not simply be minimized, it must be bounded to meet the strict requirements of these systems.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**. Providing real-time

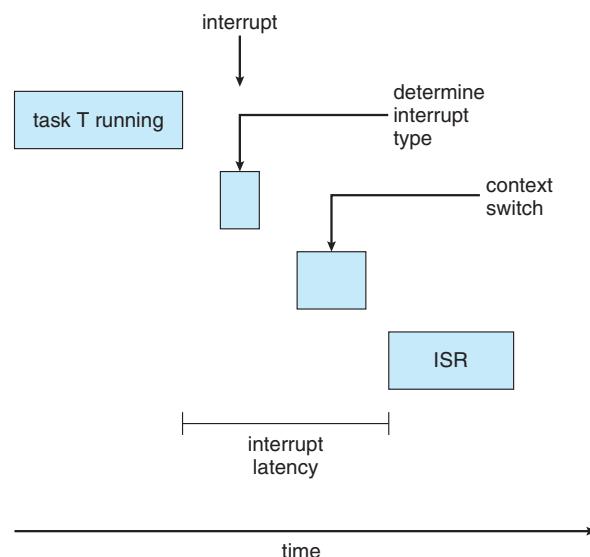


Figure 5.18 Interrupt latency.

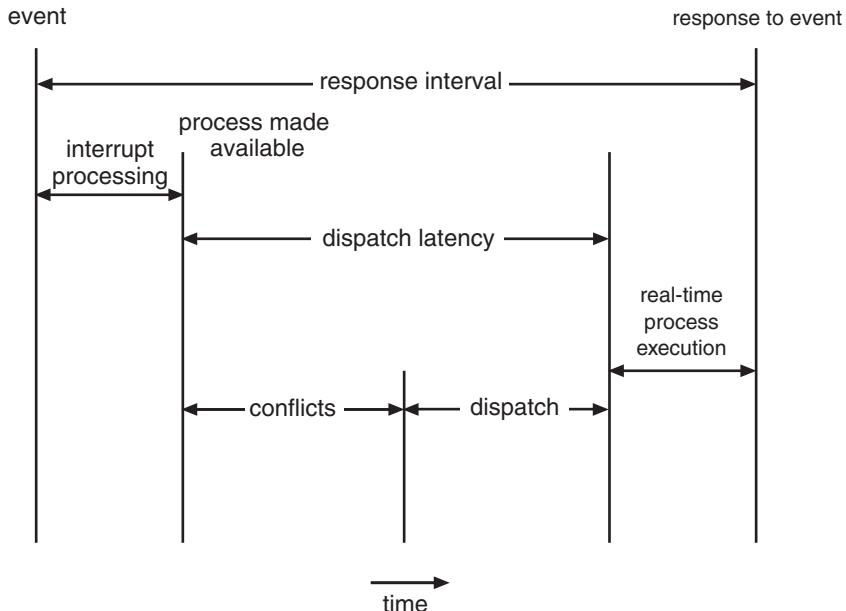


Figure 5.19 Dispatch latency.

tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency as well. The most effective technique for keeping dispatch latency low is to provide preemptive kernels. For hard real-time systems, dispatch latency is typically measured in several microseconds.

In Figure 5.19, we diagram the makeup of dispatch latency. The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel
2. Release by low-priority processes of resources needed by a high-priority process

Following the conflict phase, the dispatch phase schedules the high-priority process onto an available CPU.

5.6.2 Priority-Based Scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

Preemptive, priority-based scheduling algorithms are discussed in detail in Section 5.3.4, and Section 5.7 presents examples of the soft real-time scheduling features of the Linux, Windows, and Solaris operating systems. Each of these systems assigns real-time processes the highest scheduling priority. For

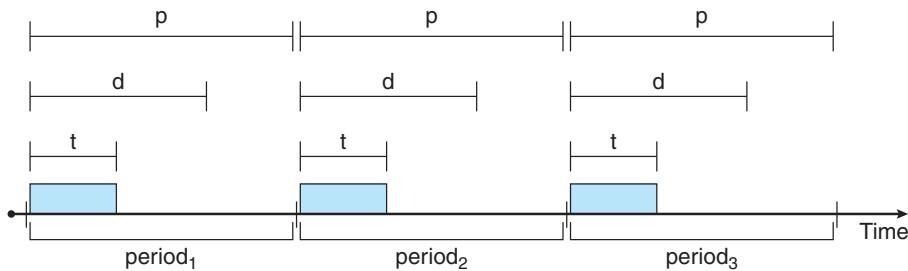


Figure 5.20 Periodic task.

example, Windows has 32 different priority levels. The highest levels—priority values 16 to 31—are reserved for real-time processes. Solaris and Linux have similar prioritization schemes.

Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features. In the remainder of this section, we cover scheduling algorithms appropriate for hard real-time systems.

Before we proceed with the details of the individual schedulers, however, we must define certain characteristics of the processes that are to be scheduled. First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods). Once a periodic process has acquired the CPU, it has a fixed processing time t , a deadline d by which it must be serviced by the CPU, and a period p . The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The **rate** of a periodic task is $1/p$. Figure 5.20 illustrates the execution of a periodic process over time. Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements.

What is unusual about this form of scheduling is that a process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control** algorithm, the scheduler does one of two things. It either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

5.6.3 Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the process-

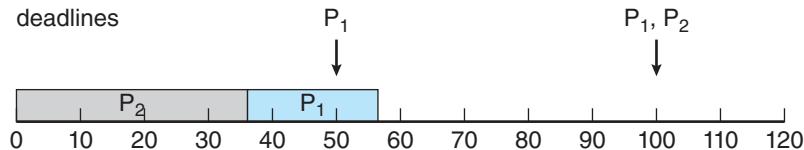


Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1 .

ing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes, P_1 and P_2 . The periods for P_1 and P_2 are 50 and 100, respectively—that is, $p_1 = 50$ and $p_2 = 100$. The processing times are $t_1 = 20$ for P_1 and $t_2 = 35$ for P_2 . The deadline for each process requires that it complete its CPU burst by the start of its next period.

We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process P_i as the ratio of its burst to its period— t_i/p_i —the CPU utilization of P_1 is $20/50 = 0.40$ and that of P_2 is $35/100 = 0.35$, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

Suppose we assign P_2 a higher priority than P_1 . The execution of P_1 and P_2 in this situation is shown in Figure 5.21. As we can see, P_2 starts execution first and completes at time 35. At this point, P_1 starts; it completes its CPU burst at time 55. However, the first deadline for P_1 was at time 50, so the scheduler has caused P_1 to miss its deadline.

Now suppose we use rate-monotonic scheduling, in which we assign P_1 a higher priority than P_2 because the period of P_1 is shorter than that of P_2 . The execution of these processes in this situation is shown in Figure 5.22. P_1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. P_2 starts running at this point and runs until time 50. At this time, it is preempted by P_1 , although it still has 5 milliseconds remaining in its CPU burst. P_1 completes its CPU burst at time 70, at which point the scheduler resumes P_2 . P_2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P_1 is scheduled again.

Rate-monotonic scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities. Let's next examine a set of processes that cannot be scheduled using the rate-monotonic algorithm.

Assume that process P_1 has a period of $p_1 = 50$ and a CPU burst of $t_1 = 25$. For P_2 , the corresponding values are $p_2 = 80$ and $t_2 = 35$. Rate-monotonic

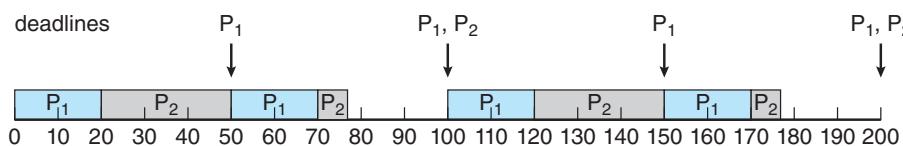


Figure 5.22 Rate-monotonic scheduling.

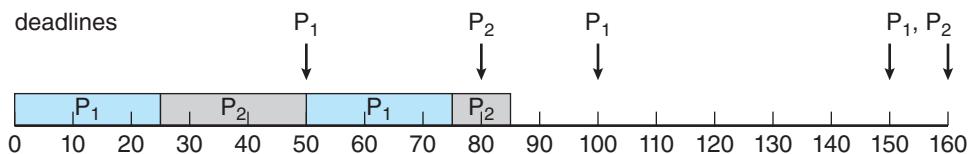


Figure 5.23 Missing deadlines with rate-monotonic scheduling.

scheduling would assign process P_1 a higher priority, as it has the shorter period. The total CPU utilization of the two processes is $(25/50) + (35/80) = 0.94$, and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 6 percent available time. Figure 5.23 shows the scheduling of processes P_1 and P_2 . Initially, P_1 runs until it completes its CPU burst at time 25. Process P_2 then begins running and runs until time 50, when it is preempted by P_1 . At this point, P_2 still has 10 milliseconds remaining in its CPU burst. Process P_1 runs until time 75; consequently, P_2 finishes its burst at time 85, after the deadline for completion of its CPU burst at time 80.

Despite being optimal, then, rate-monotonic scheduling has a limitation: CPU utilization is bounded, and it is not always possible to maximize CPU resources fully. The worst-case CPU utilization for scheduling N processes is

$$N(2^{1/N} - 1).$$

With one process in the system, CPU utilization is 100 percent, but it falls to approximately 69 percent as the number of processes approaches infinity. With two processes, CPU utilization is bounded at about 83 percent. Combined CPU utilization for the two processes scheduled in Figure 5.21 and Figure 5.22 is 75 percent; therefore, the rate-monotonic scheduling algorithm is guaranteed to schedule them so that they can meet their deadlines. For the two processes scheduled in Figure 5.23, combined CPU utilization is approximately 94 percent; therefore, rate-monotonic scheduling cannot guarantee that they can be scheduled so that they meet their deadlines.

5.6.4 Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling assigns priorities dynamically according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

To illustrate EDF scheduling, we again schedule the processes shown in Figure 5.23, which failed to meet deadline requirements under rate-monotonic scheduling. Recall that P_1 has values of $p_1 = 50$ and $t_1 = 25$ and that P_2 has values of $p_2 = 80$ and $t_2 = 35$. The EDF scheduling of these processes is shown in Figure 5.24. Process P_1 has the earliest deadline, so its initial priority is higher than that of process P_2 . Process P_2 begins running at the end of the CPU burst for P_1 . However, whereas rate-monotonic scheduling allows P_1 to preempt P_2

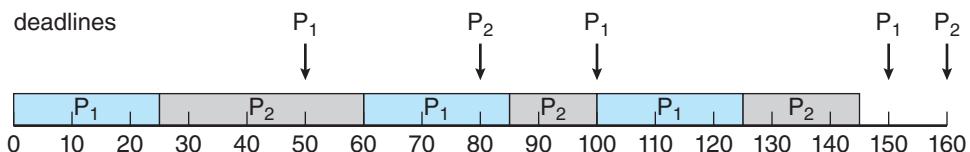


Figure 5.24 Earliest-deadline-first scheduling.

at the beginning of its next period at time 50, EDF scheduling allows process P_2 to continue running. P_2 now has a higher priority than P_1 because its next deadline (at time 80) is earlier than that of P_1 (at time 100). Thus, both P_1 and P_2 meet their first deadlines. Process P_1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. P_2 begins running at this point, only to be preempted by P_1 at the start of its next period at time 100. P_2 is preempted because P_1 has an earlier deadline (time 150) than P_2 (time 160). At time 125, P_1 completes its CPU burst and P_2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P_1 is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal—theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

5.6.5 Proportional Share Scheduling

Proportional share schedulers operate by allocating T shares among all applications. An application can receive N shares of time, thus ensuring that the application will have N/T of the total processor time. As an example, assume that a total of $T = 100$ shares is to be divided among three processes, A , B , and C . A is assigned 50 shares, B is assigned 15 shares, and C is assigned 20 shares. This scheme ensures that A will have 50 percent of total processor time, B will have 15 percent, and C will have 20 percent.

Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a client requesting a particular number of shares only if sufficient shares are available. In our current example, we have allocated $50 + 15 + 20 = 85$ shares of the total of 100 shares. If a new process D requested 30 shares, the admission controller would deny D entry into the system.

5.6.6 POSIX Real-Time Scheduling

The POSIX standard also provides extensions for real-time computing—POSIX.1b. Here, we cover some of the POSIX API related to scheduling real-time threads. POSIX defines two scheduling classes for real-time threads:

- SCHED_FIFO
- SCHED_RR

SCHED_FIFO schedules threads according to a first-come, first-served policy using a FIFO queue as outlined in Section 5.3.1. However, there is no time slicing among threads of equal priority. Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED_RR uses a round-robin policy. It is similar to SCHED_FIFO except that it provides time slicing among threads of equal priority. POSIX provides an additional scheduling class—SCHED_OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The POSIX API specifies the following two functions for getting and setting the scheduling policy:

- `pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`

The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either (1) a pointer to an integer that is set to the current scheduling policy (for `pthread_attr_getsched_policy()`) or (2) an integer value (SCHED_FIFO, SCHED_RR, or SCHED_OTHER) for the `pthread_attr_setsched_policy()` function. Both functions return nonzero values if an error occurs.

In Figure 5.25, we illustrate a POSIX Pthread program using this API. This program first determines the current scheduling policy and then sets the scheduling algorithm to SCHED_FIFO.

5.7 Operating-System Examples

We turn next to a description of the scheduling policies of the Linux, Windows, and Solaris operating systems. It is important to note that we use the term *process scheduling* in a general sense here. In fact, we are describing the scheduling of *kernel threads* with Solaris and Windows systems and of *tasks* with the Linux scheduler.

5.7.1 Example: Linux Scheduling

Process scheduling in Linux has had an interesting history. Prior to Version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm. However, as this algorithm was not designed with SMP systems in mind, it did not adequately support systems with multiple processors. In addition, it resulted in poor performance for systems with a large number of runnable processes. With Version 2.5 of the kernel, the scheduler was overhauled to include a scheduling algorithm—known as *O(1)*—that ran in constant time regardless of the number of tasks in the system. The *O(1)* scheduler also provided

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    /* set the scheduling policy - FIFO, RR, or OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
        fprintf(stderr, "Unable to set policy.\n");

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */

    pthread_exit(0);
}
```

Figure 5.25 POSIX real-time scheduling API.

increased support for SMP systems, including processor affinity and load balancing between processors. However, in practice, although the $O(1)$ scheduler delivered excellent performance on SMP systems, it led to poor response times for the interactive processes that are common on many desktop computer systems. During development of the 2.6 kernel, the scheduler was again revised; and in release 2.6.23 of the kernel, the *Completely Fair Scheduler* (CFS) became the default Linux scheduling algorithm.

Scheduling in the Linux system is based on **scheduling classes**. Each class is assigned a specific priority. By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes. The scheduling criteria for a Linux server, for example, may be different from those for a mobile device running Linux. To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class. Standard Linux kernels implement two scheduling classes: (1) a default scheduling class using the CFS scheduling algorithm and (2) a real-time scheduling class. We discuss each of these classes here. New scheduling classes can, of course, be added.

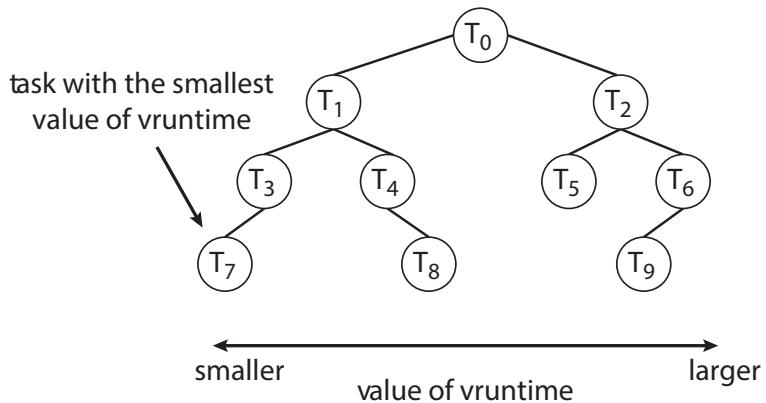
Rather than using strict rules that associate a relative priority value with the length of a time quantum, the CFS scheduler assigns a proportion of CPU processing time to each task. This proportion is calculated based on the **nice value** assigned to each task. Nice values range from -20 to $+19$, where a numerically lower nice value indicates a higher relative priority. Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values. The default nice value is 0 . (The term *nice* comes from the idea that if a task increases its nice value from, say, 0 to $+10$, it is being nice to other tasks in the system by lowering its relative priority. In other words, nice processes finish last!) CFS doesn't use discrete values of time slices and instead identifies a **targeted latency**, which is an interval of time during which every runnable task should run at least once. Proportions of CPU time are allocated from the value of targeted latency. In addition to having default and minimum values, targeted latency can increase if the number of active tasks in the system grows beyond a certain threshold.

The CFS scheduler doesn't directly assign priorities. Rather, it records how long each task has run by maintaining the **virtual run time** of each task using the per-task variable `vruntime`. The virtual run time is associated with a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks. For tasks at normal priority (nice values of 0), virtual run time is identical to actual physical run time. Thus, if a task with default priority runs for 200 milliseconds, its `vruntime` will also be 200 milliseconds. However, if a lower-priority task runs for 200 milliseconds, its `vruntime` will be higher than 200 milliseconds. Similarly, if a higher-priority task runs for 200 milliseconds, its `vruntime` will be less than 200 milliseconds. To decide which task to run next, the scheduler simply selects the task that has the smallest `vruntime` value. In addition, a higher-priority task that becomes available to run can preempt a lower-priority task.

Let's examine the CFS scheduler in action: Assume that two tasks have the same nice values. One task is I/O-bound, and the other is CPU-bound. Typically, the I/O-bound task will run only for short periods before blocking for additional I/O, and the CPU-bound task will exhaust its time period whenever it has an opportunity to run on a processor. Therefore, the value of `vruntime` will

CFS PERFORMANCE

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Rather than using a standard queue data structure, each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below.



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\log N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

eventually be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task. At that point, if the CPU-bound task is executing when the I/O-bound task becomes eligible to run (for example, when I/O the task is waiting for becomes available), the I/O-bound task will preempt the CPU-bound task.

Linux also implements real-time scheduling using the POSIX standard as described in Section 5.6.6. Any task scheduled using either the `SCHED_FIFO` or the `SCHED_RR` real-time policy runs at a higher priority than normal (non-real-time) tasks. Linux uses two separate priority ranges, one for real-time tasks and a second for normal tasks. Real-time tasks are assigned static priorities within the range of 0 to 99, and normal tasks are assigned priorities from 100 to 139. These two ranges map into a global priority scheme wherein numerically lower values indicate higher relative priorities. Normal tasks are assigned a priority

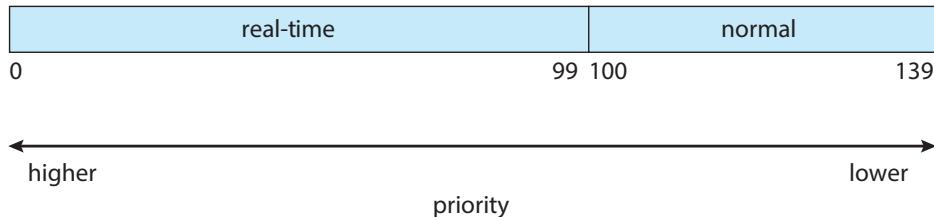


Figure 5.26 Scheduling priorities on a Linux system.

based on their nice values, where a value of -20 maps to priority 100 and a nice value of $+19$ maps to 139. This scheme is shown in Figure 5.26.

The CFS scheduler also supports load balancing, using a sophisticated technique that equalizes the load among processing cores yet is also NUMA-aware and minimizes the migration of threads. CFS defines the load of each thread as a combination of the thread's priority and its average rate of CPU utilization. Therefore, a thread that has a high priority, yet is mostly I/O-bound and requires little CPU usage, has a generally low load, similar to the load of a low-priority thread that has high CPU utilization. Using this metric, the load of a queue is the sum of the loads of all threads in the queue, and balancing is simply ensuring that all queues have approximately the same load.

As highlighted in Section 5.5.4, however, migrating a thread may result in a memory access penalty due to either having to invalidate cache contents or, on NUMA systems, incurring longer memory access times. To address this problem, Linux identifies a hierarchical system of scheduling domains. A **scheduling domain** is a set of CPU cores that can be balanced against one another. This idea is illustrated in Figure 5.27. The cores in each scheduling domain are grouped according to how they share the resources of the system. For example, although each core shown in Figure 5.27 may have its own level 1 (L1) cache, pairs of cores share a level 2 (L2) cache and are thus organized into separate $domain_0$ and $domain_1$. Likewise, these two domains may share a level 3 (L3) cache, and are therefore organized into a processor-level domain (also known as a **NUMA node**). Taking this one-step further, on a NUMA system,

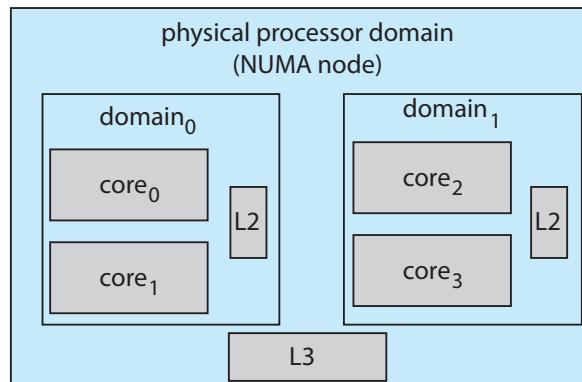


Figure 5.27 NUMA-aware load balancing with Linux CFS scheduler.

a larger system-level domain would combine separate processor-level NUMA nodes.

The general strategy behind CFS is to balance loads within domains, beginning at the lowest level of the hierarchy. Using Figure 5.27 as an example, initially a thread would only migrate between cores on the same domain (i.e. within $domain_0$ or $domain_1$.) Load balancing at the next level would occur between $domain_0$ and $domain_1$. CFS is reluctant to migrate threads between separate NUMA nodes if a thread would be moved farther from its local memory, and such migration would only occur under severe load imbalances. As a general rule, if the overall system is busy, CFS will not load-balance beyond the domain local to each core to avoid the memory latency penalties of NUMA systems.

5.7.2 Example: Windows Scheduling

Windows schedules threads using a priority-based, preemptive scheduling algorithm. The Windows scheduler ensures that the highest-priority thread will always run. The portion of the Windows kernel that handles scheduling is called the **dispatcher**. A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes. The **variable class** contains threads having priorities from 1 to 15, and the **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.) The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the **idle thread**.

There is a relationship between the numeric priorities of the Windows kernel and the Windows API. The Windows API identifies the following six priority classes to which a process can belong:

- IDLE_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- REALTIME_PRIORITY_CLASS

Processes are typically members of the NORMAL_PRIORITY_CLASS. A process belongs to this class unless the parent of the process was a member of the IDLE_PRIORITY_CLASS or unless another class was specified when the process was created. Additionally, the priority class of a process can be altered with

the `SetPriorityClass()` function in the Windows API. Priorities in all classes except the `REALTIME_PRIORITY_CLASS` are variable, meaning that the priority of a thread belonging to one of these classes can change.

A thread within a given priority class also has a relative priority. The values for relative priorities include:

- IDLE
- LOWEST
- BELOW_NORMAL
- NORMAL
- ABOVE_NORMAL
- HIGHEST
- TIME_CRITICAL

The priority of each thread is based on both the priority class it belongs to and its relative priority within that class. This relationship is shown in Figure 5.28. The values of the priority classes appear in the top row. The left column contains the values for the relative priorities. For example, if the relative priority of a thread in the `ABOVE_NORMAL_PRIORITY_CLASS` is `NORMAL`, the numeric priority of that thread is 10.

Furthermore, each thread has a base priority representing a value in the priority range for the class to which the thread belongs. By default, the base priority is the value of the `NORMAL` relative priority for that class. The base priorities for each priority class are as follows:

- `REALTIME_PRIORITY_CLASS`—24
- `HIGH_PRIORITY_CLASS`—13
- `ABOVE_NORMAL_PRIORITY_CLASS`—10
- `NORMAL_PRIORITY_CLASS`—8

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 5.28 Windows thread priorities.

- BELOW_NORMAL_PRIORITY_CLASS—6
- IDLE_PRIORITY_CLASS—4

The initial priority of a thread is typically the base priority of the process the thread belongs to, although the `SetThreadPriority()` function in the Windows API can also be used to modify a thread's base priority.

When a thread's time quantum runs out, that thread is interrupted. If the thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority, however. Lowering the priority tends to limit the CPU consumption of compute-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on what the thread was waiting for. For example, a thread waiting for keyboard I/O would get a large increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads that are using the mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background. This strategy is used by several operating systems, including UNIX. In addition, the window with which the user is currently interacting receives a priority boost to enhance its response time.

When a user is running an interactive program, the system needs to provide especially good performance. For this reason, Windows has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS. Windows distinguishes between the **foreground process** that is currently selected on the screen and the **background processes** that are not currently selected. When a process moves into the foreground, Windows increases the scheduling quantum by some factor—typically by 3. This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

Windows 7 introduced **user-mode scheduling (UMS)**, which allows applications to create and manage threads independently of the kernel. Thus, an application can create and schedule multiple threads without involving the Windows kernel scheduler. For applications that create a large number of threads, scheduling threads in user mode is much more efficient than kernel-mode thread scheduling, as no kernel intervention is necessary.

Earlier versions of Windows provided a similar feature known as **fiber**, which allowed several user-mode threads (fibers) to be mapped to a single kernel thread. However, fibers were of limited practical use. A fiber was unable to make calls to the Windows API because all fibers had to share the thread environment block (TEB) of the thread on which they were running. This presented a problem if a Windows API function placed state information into the TEB for one fiber, only to have the information overwritten by a different fiber. UMS overcomes this obstacle by providing each user-mode thread with its own thread context.

In addition, unlike fibers, UMS is not intended to be used directly by the programmer. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. For example, Microsoft provides **Concurrency Runtime** (ConCRT), a concurrent programming framework for C++ that is designed for task-based parallelism

(Section 4.2) on multicore processors. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available processing cores.

Windows also supports scheduling on multiprocessor systems as described in Section 5.5 by attempting to schedule a thread on the most optimal processing core for that thread, which includes maintaining a thread's preferred as well as most recent processor. One technique used by Windows is to create sets of logical processors (known as **SMT sets**). On a hyper-threaded SMT system, hardware threads belonging to the same CPU core would also belong to the same SMT set. Logical processors are numbered, beginning from 0. As an example, a dual-threaded/quad-core system would contain eight logical processors, consisting of the four SMT sets: {0, 1}, {2, 3}, {4, 5}, and {6, 7}. To avoid cache memory access penalties highlighted in Section 5.5.4, the scheduler attempts to maintain a thread running on logical processors within the same SMT set.

To distribute loads across different logical processors, each thread is assigned an **ideal processor**, which is a number representing a thread's preferred processor. Each process has an initial seed value identifying the ideal CPU for a thread belonging to that process. This seed is incremented for each new thread created by that process, thereby distributing the load across different logical processors. On SMT systems, the increment for the next ideal processor is in the next SMT set. For example, on a dual-threaded/quad-core system, the ideal processors for threads in a specific process would be assigned 0, 2, 4, 6, 0, 2, To avoid the situation whereby the first thread for each process is assigned processor 0, processes are assigned different seed values, thereby distributing the load of threads across all physical processing cores in the system. Continuing our example from above, if the seed for a second process were 1, the ideal processors would be assigned in the order 1, 3, 5, 7, 1, 3, and so forth.

5.7.3 Example: Solaris Scheduling

Solaris uses priority-based thread scheduling. Each thread belongs to one of six classes:

1. Time sharing (TS)
2. Interactive (IA)
3. Real time (RT)
4. System (SYS)
5. Fair share (FSS)
6. Fixed priority (FP)

Within each class there are different priorities and different scheduling algorithms.

The default scheduling class for a process is time sharing. The scheduling policy for the time-sharing class dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue. By default, there is an inverse relationship between priorities and time slices. The higher the

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 5.29 Solaris dispatch table for time-sharing and interactive threads.

priority, the smaller the time slice; and the lower the priority, the larger the time slice. Interactive processes typically have a higher priority; CPU-bound processes, a lower priority. This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes. The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications—such as those created by the KDE or GNOME window managers—a higher priority for better performance.

Figure 5.29 shows the simplified dispatch table for scheduling time-sharing and interactive threads. These two scheduling classes include 60 priority levels, but for brevity, we display only a handful. (To see the full dispatch table on a Solaris system or VM, run `dispadmin -c TS -g`.) The dispatch table shown in Figure 5.29 contains the following fields:

- **Priority.** The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.
- **Time quantum.** The time quantum for the associated priority. This illustrates the inverse relationship between priorities and time quanta: the lowest priority (priority 0) has the highest time quantum (200 milliseconds), and the highest priority (priority 59) has the lowest time quantum (20 milliseconds).
- **Time quantum expired.** The new priority of a thread that has used its entire time quantum without blocking. Such threads are considered CPU-intensive. As shown in the table, these threads have their priorities lowered.

- **Return from sleep.** The priority of a thread that is returning from sleeping (such as from waiting for I/O). As the table illustrates, when I/O is available for a waiting thread, its priority is boosted to between 50 and 59, supporting the scheduling policy of providing good response time for interactive processes.

Threads in the real-time class are given the highest priority. A real-time process will run before a process in any other class. This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time. In general, however, few processes belong to the real-time class.

Solaris uses the system class to run kernel threads, such as the scheduler and paging daemon. Once the priority of a system thread is established, it does not change. The system class is reserved for kernel use (user processes running in kernel mode are not in the system class).

The fixed-priority and fair-share classes were introduced with Solaris 9. Threads in the fixed-priority class have the same priority range as those in the time-sharing class; however, their priorities are not dynamically adjusted. The fair-share class uses CPU **shares** instead of priorities to make scheduling decisions. CPU shares indicate entitlement to available CPU resources and are allocated to a set of processes (known as a **project**).

Each scheduling class includes a set of priorities. However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run. The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread. If there are multiple threads with the same priority, the scheduler uses a round-robin queue. Figure 5.30 illustrates how the six scheduling classes relate to one another and how they map to global priorities. Notice that the kernel maintains ten threads for servicing interrupts. These threads do not belong to any scheduling class and execute at the highest priority (160–169). As mentioned, Solaris has traditionally used the many-to-many model (Section 4.3.3) but switched to the one-to-one model (Section 4.3.2) beginning with Solaris 9.

5.8 Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system? As we saw in Section 5.3, there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.

The first problem is defining the criteria to be used in selecting an algorithm. As we saw in Section 5.2, criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these elements. Our criteria may include several measures, such as these:

- Maximizing CPU utilization under the constraint that the maximum response time is 300 milliseconds

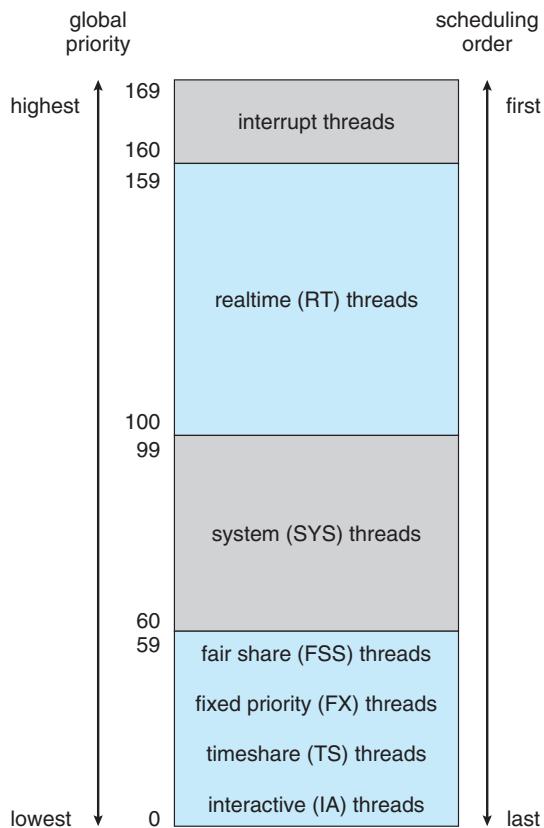


Figure 5.30 Solaris scheduling.

- Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.

5.8.1 Deterministic Modeling

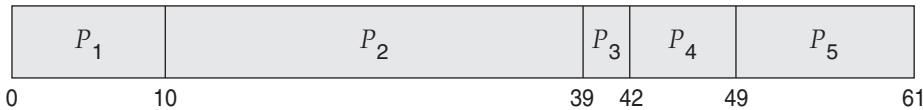
One major class of evaluation methods is **analytic evaluation**. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for that workload.

Deterministic modeling is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

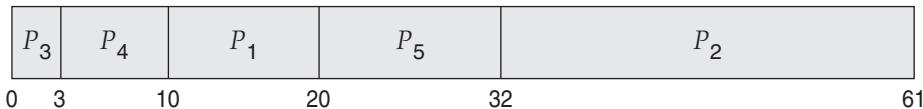
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



The waiting time is 0 milliseconds for process P_1 , 10 milliseconds for process P_2 , 39 milliseconds for process P_3 , 42 milliseconds for process P_4 , and 49 milliseconds for process P_5 . Thus, the average waiting time is $(0 + 10 + 39 + 42 + 49)/5 = 28$ milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as



The waiting time is 10 milliseconds for process P_1 , 32 milliseconds for process P_2 , 0 milliseconds for process P_3 , 3 milliseconds for process P_4 , and 20 milliseconds for process P_5 . Thus, the average waiting time is $(10 + 32 + 0 + 3 + 20)/5 = 13$ milliseconds.

With the RR algorithm, we execute the processes as



The waiting time is 0 milliseconds for process P_1 , 32 milliseconds for process P_2 , 20 milliseconds for process P_3 , 23 milliseconds for process P_4 , and 40 milliseconds for process P_5 . Thus, the average waiting time is $(0 + 32 + 20 + 23 + 40)/5 = 23$ milliseconds.

We can see that, in this case, the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can

measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

5.8.2 Queueing Models

On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or simply estimated. The result is a mathematical formula describing the probability of a particular CPU burst. Commonly, this distribution is exponential and is described by its mean. Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis**.

As an example, let n be the average long-term queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W.$$

This equation, known as **Little's formula**, is particularly useful because it is valid for any scheduling algorithm and arrival distribution. For example n could be the number of customers in a store.

We can use Little's formula to compute one of the three variables if we know the other two. For example, if we know that 7 processes arrive every second (on average) and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds.

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Thus, arrival and service distributions are often defined in mathematically tractable—but unrealistic—ways. It is also generally necessary to make a number of independent assumptions, which may not be accurate. As a result of these difficulties, queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

5.8.3 Simulations

To get a more accurate evaluation of scheduling algorithms, we can use simulations. Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock. As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions. The distributions can be defined mathematically (uniform, exponential, Poisson) or empirically. If a distribution is to be defined empirically, measurements of the actual system under study are taken. The results define the distribution of events in the real system; this distribution can then be used to drive the simulation.

A distribution-driven simulation may be inaccurate, however, because of relationships between successive events in the real system. The frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use **trace files**. We create a trace by monitoring the real system and recording the sequence of actual events (Figure 5.31). We then use this sequence to drive the simulation. Trace files provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulations can be expensive, often requiring many hours of computer time. A more detailed simulation provides more accurate results, but it also

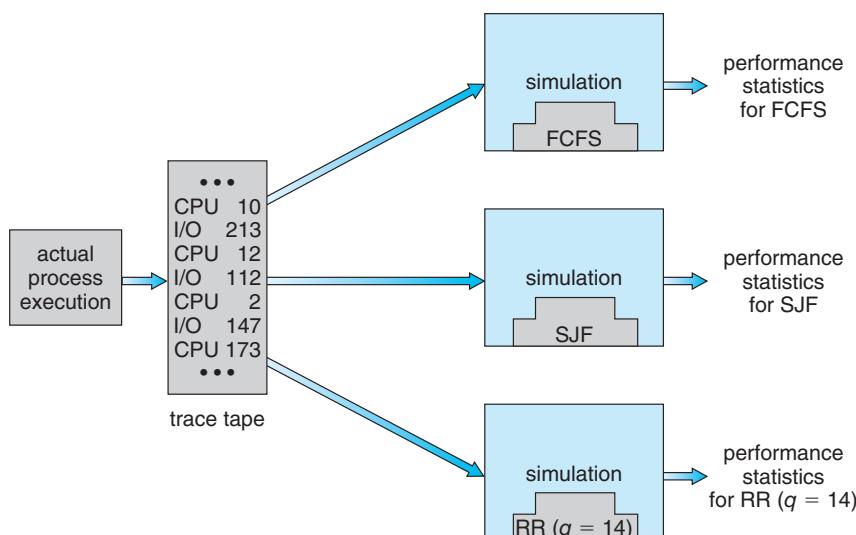


Figure 5.31 Evaluation of CPU schedulers by simulation.

takes more computer time. In addition, trace files can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

5.8.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

This method is not without expense. The expense is incurred in coding the algorithm and modifying the operating system to support it (along with its required data structures). There is also cost in testing the changes, usually in virtual machines rather than on dedicated hardware. **Regression testing** confirms that the changes haven't made anything worse, and haven't caused new bugs or caused old bugs to be recreated (for example because the algorithm being replaced solved some bug and changing it caused that bug to reoccur).

Another difficulty is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use. This problem is usually addressed by using tools or scripts that encapsulate complete sets of actions, repeatedly using those tools, and using those tools while measuring the results (and detecting any problems they cause in the new environment).

Of course human or program behavior can attempt to circumvent scheduling algorithms. For example, researchers designed one system that classified interactive and noninteractive processes automatically by looking at the amount of terminal I/O. If a process did not input or output to the terminal in a 1-second interval, the process was classified as noninteractive and was moved to a lower-priority queue. In response to this policy, one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 second. The system gave his programs a high priority, even though the terminal output was completely meaningless.

In general, most flexible scheduling algorithms are those that can be altered by the system managers or by the users so that they can be tuned for a specific application or set of applications. A workstation that performs high-end graphical applications, for instance, may have scheduling needs different from those of a web server or file server. Some operating systems—particularly several versions of UNIX—allow the system manager to fine-tune the scheduling parameters for a particular system configuration. For example, Solaris provides the `dispadmin` command to allow the system administrator to modify the parameters of the scheduling classes described in Section 5.7.3.

Another approach is to use APIs that can modify the priority of a process or thread. The Java, POSIX, and Windows APIs provide such functions. The downfall of this approach is that performance-tuning a system or application most often does not result in improved performance in more general situations.

5.9 Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- Scheduling algorithms may be either preemptive (where the CPU can be taken away from a process) or nonpreemptive (where a process must voluntarily relinquish control of the CPU). Almost all modern operating systems are preemptive.
- Scheduling algorithms can be evaluated according to the following five criteria: (1) CPU utilization, (2) throughput, (3) turnaround time, (4) waiting time, and (5) response time.
- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.
- Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult.
- Round-robin (RR) scheduling allocates the CPU to each process for a time quantum. If the process does not relinquish the CPU before its time quantum expires, the process is preempted, and another process is scheduled to run for a time quantum.
- Priority scheduling assigns each process a priority, and the CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling.
- Multilevel queue scheduling partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.
- Multilevel feedback queues are similar to multilevel queues, except that a process may migrate between different queues.
- Multicore processors place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU.
- Load balancing on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.
- Soft real-time scheduling gives priority to real-time tasks over non-real-time tasks. Hard real-time scheduling provides timing guarantees for real-time tasks,
- Rate-monotonic real-time scheduling schedules periodic tasks using a static priority policy with preemption.

- Earliest-deadline-first (EDF) scheduling assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.
- Proportional share scheduling allocates T shares among all applications. If an application is allocated N shares of time, it is ensured of having N/T of the total processor time.
- Linux uses the completely fair scheduler (CFS), which assigns a proportion of CPU processing time to each task. The proportion is based on the virtual runtime (`vruntime`) value associated with each task.
- Windows scheduling uses a preemptive, 32-level priority scheme to determine the order of thread scheduling.
- Solaris identifies six unique scheduling classes that are mapped to a global priority. CPU-intensive threads are generally assigned lower priorities (and longer time quanta), and I/O-bound threads are usually assigned higher priorities (with shorter time quanta.)
- Modeling and simulations can be used to evaluate a CPU scheduling algorithm.

Practice Exercises

- 5.1 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n .
- 5.2 Explain the difference between preemptive and nonpreemptive scheduling.
- 5.3 Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

- a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?
- b. What is the average turnaround time for these processes with the SJF scheduling algorithm?
- c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the

average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as *future-knowledge scheduling*.

- 5.4 Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5 , all at time 0.

- a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).
 - b. What is the turnaround time of each process for each of the scheduling algorithms in part a?
 - c. What is the waiting time of each process for each of these scheduling algorithms?
 - d. Which of the algorithms results in the minimum average waiting time (over all processes)?
- 5.5 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm.

<u>Process</u>	<u>Priority</u>	<u>Burst</u>	<u>Arrival</u>
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an **idle task** (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a

- time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.
- Show the scheduling order of the processes using a Gantt chart.
 - What is the turnaround time for each process?
 - What is the waiting time for each process?
 - What is the CPU utilization rate?
- 5.6** What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?
- 5.7** Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.
- These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?
- Priority and SJF
 - Multilevel feedback queues and FCFS
 - Priority and FCFS
 - RR and SJF
- 5.8** Suppose that a CPU scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?
- 5.9** Distinguish between PCS and SCS scheduling.
- 5.10** The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

$$\text{Priority} = (\text{recent CPU usage} / 2) + \text{base}$$

where $\text{base} = 60$ and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P_1 is 40, for process P_2 is 18, and for process P_3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Further Reading

Scheduling policies used in the UNIX FreeBSD 5.2 are presented by [McKusick et al. (2015)]; The Linux CFS scheduler is further described in <https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/>. Solaris scheduling is described by [Mauro and McDougall (2007)]. [Russinovich et al. (2017)] discusses scheduling in Windows internals. [Butenhof (1997)] and [Lewis and Berg (1998)] describe scheduling in Pthreads systems. Multicore scheduling is examined in [McNairy and Bhatia (2005)], [Kongetira et al. (2005)], and [Siddha et al. (2007)] .

Bibliography

- [**Butenhof (1997)**] D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley (1997).
- [**Kongetira et al. (2005)**] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-Way Multithreaded SPARC Processor”, *IEEE Micro Magazine*, Volume 25, Number 2 (2005), pages 21–29.
- [**Lewis and Berg (1998)**] B. Lewis and D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press (1998).
- [**Mauro and McDougall (2007)**] J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2007).
- [**McKusick et al. (2015)**] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System—Second Edition*, Pearson (2015).
- [**McNairy and Bhatia (2005)**] C. McNairy and R. Bhatia, “Montecito: A Dual-Core, Dual-Threaded Itanium Processor”, *IEEE Micro Magazine*, Volume 25, Number 2 (2005), pages 10–20.
- [**Russinovich et al. (2017)**] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).
- [**Siddha et al. (2007)**] S. Siddha, V. Pallipadi, and A. Mallick, “Process Scheduling Challenges in the Era of Multi-Core Processors”, *Intel Technology Journal*, Volume 11, Number 4 (2007).

Chapter 5 Exercises

- 5.11 Of these two types of programs:

- a. I/O-bound
- b. CPU-bound

which is more likely to have voluntary context switches, and which is more likely to have nonvoluntary context switches? Explain your answer.

- 5.12 Discuss how the following pairs of scheduling criteria conflict in certain settings.

- a. CPU utilization and response time
- b. Average turnaround time and maximum waiting time
- c. I/O device utilization and CPU utilization

- 5.13 One technique for implementing **lottery scheduling** works by assigning processes lottery tickets, which are used for allocating CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the CPU. The BTV operating system implements lottery scheduling by holding a lottery 50 times each second, with each lottery winner getting 20 milliseconds of CPU time ($20 \text{ milliseconds} \times 50 = 1 \text{ second}$). Describe how the BTV scheduler can ensure that higher-priority threads receive more attention from the CPU than lower-priority threads.

- 5.14 Most scheduling algorithms maintain a **run queue**, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. What are the advantages and disadvantages of each of these approaches?

- 5.15 Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

- a. $\alpha = 0$ and $\tau_0 = 100 \text{ milliseconds}$
- b. $\alpha = 0.99$ and $\tau_0 = 10 \text{ milliseconds}$

- 5.16 A variation of the round-robin scheduler is the **regressive round-robin** scheduler. This scheduler assigns each process a time quantum and a priority. The initial value of a time quantum is 50 milliseconds. However, every time a process has been allocated the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted. (The time quantum for a process can be increased to a maximum of 100 milliseconds.) When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the same. What type of process (CPU-bound or I/O-bound) does the regressive round-robin scheduler favor? Explain.

- 5.17** Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	5	4
P_2	3	1
P_3	1	2
P_4	7	2
P_5	4	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5 , all at time 0.

- a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).
 - b. What is the turnaround time of each process for each of the scheduling algorithms in part a?
 - c. What is the waiting time of each process for each of these scheduling algorithms?
 - d. Which of the algorithms results in the minimum average waiting time (over all processes)?
- 5.18** The following processes are being scheduled using a preemptive, priority-based, round-robin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
P_6	5	15	55

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. The scheduler will execute the highest-priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

- a. Show the scheduling order of the processes using a Gantt chart.
 - b. What is the turnaround time for each process?
 - c. What is the waiting time for each process?
- 5.19** The `nice` command is used to set the nice value of a process on Linux, as well as on other UNIX systems. Explain why some systems may allow any user to assign a process a nice value ≥ 0 yet allow only the root (or administrator) user to assign nice values < 0 .

- 5.20 Which of the following scheduling algorithms could result in starvation?
- First-come, first-served
 - Shortest job first
 - Round robin
 - Priority
- 5.21 Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.
- What would be the effect of putting two pointers to the same process in the ready queue?
 - What would be two major advantages and two disadvantages of this scheme?
 - How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?
- 5.22 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:
- The time quantum is 1 millisecond
 - The time quantum is 10 milliseconds
- 5.23 Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?
- 5.24 Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α . When it is running, its priority changes at a rate β . All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.
- What is the algorithm that results from $\beta > \alpha > 0$?
 - What is the algorithm that results from $\alpha < \beta < 0$?
- 5.25 Explain how the following scheduling algorithms discriminate either in favor of or against short processes:
- FCFS
 - RR
 - Multilevel feedback queues

- 5.26 Describe why a shared ready queue might suffer from performance problems in an SMP environment.
- 5.27 Consider a load-balancing algorithm that ensures that each queue has approximately the same number of threads, independent of priority. How effectively would a priority-based scheduling algorithm handle this situation if one run queue had all high-priority threads and a second queue had all low-priority threads?
- 5.28 Assume that an SMP system has private, per-processor run queues. When a new process is created, it can be placed in either the same queue as the parent process or a separate queue.
- What are the benefits of placing the new process in the same queue as its parent?
 - What are the benefits of placing the new process in a different queue?
- 5.29 Assume that a thread has blocked for network I/O and is eligible to run again. Describe why a NUMA-aware scheduling algorithm should reschedule the thread on the same CPU on which it previously ran.
- 5.30 Using the Windows scheduling algorithm, determine the numeric priority of each of the following threads.
- A thread in the REALTIME_PRIORITY_CLASS with a relative priority of NORMAL
 - A thread in the ABOVE_NORMAL_PRIORITY_CLASS with a relative priority of HIGHEST
 - A thread in the BELOW_NORMAL_PRIORITY_CLASS with a relative priority of ABOVE_NORMAL
- 5.31 Assuming that no threads belong to the REALTIME_PRIORITY_CLASS and that none may be assigned a TIME_CRITICAL priority, what combination of priority class and priority corresponds to the highest possible relative priority in Windows scheduling?
- 5.32 Consider the scheduling algorithm in the Solaris operating system for time-sharing threads.
- What is the time quantum (in milliseconds) for a thread with priority 15? With priority 40?
 - Assume that a thread with priority 50 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?
 - Assume that a thread with priority 20 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

- 5.33 Assume that two tasks, A and B , are running on a Linux system. The nice values of A and B are -5 and $+5$, respectively. Using the CFS scheduler as a guide, describe how the respective values of vruntime vary between the two processes given each of the following scenarios:
- Both A and B are CPU-bound.
 - A is I/O-bound, and B is CPU-bound.
 - A is CPU-bound, and B is I/O-bound.
- 5.34 Provide a specific circumstance that illustrates where rate-monotonic scheduling is inferior to earliest-deadline-first scheduling in meeting real-time process deadlines?
- 5.35 Consider two processes, P_1 and P_2 , where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.
- a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart such as the ones in Figure 5.21–Figure 5.24.
 - b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.
- 5.36 Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.
- 5.37 Describe the advantages of using heterogeneous multiprocessing in a mobile system.

Programming Projects

Scheduling Algorithms

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

- First-come, first-served (FCFS), which schedules tasks in the order in which they request the CPU.
- Shortest-job-first (SJF), which schedules tasks in order of the length of the tasks' next CPU burst.
- Priority scheduling, which schedules tasks based on priority.
- Round-robin (RR) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst).
- Priority with round-robin, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is 10 milliseconds.

I. Implementation

The implementation of this project may be completed in either C or Java, and program files supporting both of these languages are provided in the source code download for the text. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form [*task name*] [*priority*] [*CPU burst*], with the following example format:

```
T1, 4, 20  
T2, 2, 25  
T3, 3, 25  
T4, 3, 15  
T5, 10, 10
```

Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that all tasks arrive at the same time, so your scheduler algorithms do not have to support higher-priority processes preempting processes with lower priorities. In addition, tasks do not have to be placed into a queue or list in any particular order.

There are a few different strategies for organizing the list of tasks, as first presented in Section 5.1.2. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling

algorithm. For example, SJF scheduling would search the list to find the task with the shortest next CPU burst. Alternatively, a list could be ordered according to scheduling criteria (that is, by priority). One other strategy involves having a separate queue for each unique priority, as shown in Figure 5.7. These approaches are briefly discussed in Section 5.3.6. It is also worth highlighting that we are using the terms *list* and *queue* somewhat interchangeably. However, a queue has very specific FIFO functionality, whereas a list does not have such strict insertion and deletion requirements. You are likely to find the functionality of a general list to be more suitable when completing this project.

II. C Implementation Details

The file `driver.c` reads in the schedule of tasks, inserts each task into a linked list, and invokes the process scheduler by calling the `schedule()` function. The `schedule()` function executes each task according to the specified scheduling algorithm. Tasks selected for execution on the CPU are determined by the `pickNextTask()` function and are executed by invoking the `run()` function defined in the `CPU.c` file. A `Makefile` is used to determine the specific scheduling algorithm that will be invoked by `driver`. For example, to build the FCFS scheduler, we would enter

```
make fcfs
```

and would execute the scheduler (using the schedule of tasks `schedule.txt`) as follows:

```
./fcfs schedule.txt
```

Refer to the `README` file in the source code download for further details. Before proceeding, be sure to familiarize yourself with the source code provided as well as the `Makefile`.

III. Java Implementation Details

The file `Driver.java` reads in the schedule of tasks, inserts each task into a Java `ArrayList`, and invokes the process scheduler by calling the `schedule()` method. The following interface identifies a generic scheduling algorithm, which the five different scheduling algorithms will implement:

```
public interface Algorithm
{
    // Implementation of scheduling algorithm
    public void schedule();

    // Selects the next task to be scheduled
    public Task pickNextTask();
}
```

The `schedule()` method obtains the next task to be run on the CPU by invoking the `pickNextTask()` method and then executes this `Task` by calling the static `run()` method in the `CPU.java` class.

The program is run as follows:

```
java Driver fcfs schedule.txt
```

Refer to the README file in the source code download for further details. Before proceeding, be sure to familiarize yourself with all Java source files provided in the source code download.

IV. Further Challenges

Two additional challenges are presented for this project:

1. Each task provided to the scheduler is assigned a unique task (tid). If a scheduler is running in an SMP environment where each CPU is separately running its own scheduler, there is a possible race condition on the variable that is used to assign task identifiers. Fix this race condition using an atomic integer.

On Linux and macOS systems, the `__sync_fetch_and_add()` function can be used to atomically increment an integer value. As an example, the following code sample atomically increments `value` by 1:

```
int value = 0;  
__sync_fetch_and_add(&value, 1);
```

Refer to the Java API for details on how to use the `AtomicInteger` class for Java programs.

2. Calculate the average turnaround time, waiting time, and response time for each of the scheduling algorithms.

Part Three

Process Synchronization

A system typically consists of several (perhaps hundreds or even thousands) of threads running either concurrently or in parallel. Threads often share user data. Meanwhile, the operating system continuously updates various data structures to support multiple threads. A *race condition* exists when access to shared data is not controlled, possibly resulting in corrupt data values.

Process synchronization involves using tools that control access to shared data to avoid race conditions. These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock.

Synchronization Tools



A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

CHAPTER OBJECTIVES

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

6.1 Background

We've already seen that processes can execute concurrently or in parallel. Section 3.2.2 introduced the role of process scheduling and described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process. Additionally, Section 4.2 introduced parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores. In this chapter, we explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

Let's consider an example of how this can happen. In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer–consumer problem, which is a representative paradigm of many operating system functions. Specifically, in Section 3.5, we described how a bounded buffer could be used to enable processes to share memory.

We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at most BUFFER_SIZE – 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable, count, initialized to 0. count is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```

while (true) {
    /* produce an item in next_produced */

    while (count == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

```

The code for the consumer process can be modified as follows:

```

while (true) {
    while (count == 0)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    /* consume the item in next_consumed */
}

```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable count is currently 5 and that the producer and consumer processes concurrently execute the statements “count++” and “count–”. Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! The only correct result, though, is count == 5, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `count` may be incorrect as follows. Note that the statement “`count++`” may be implemented in machine language (on a typical machine) as follows:

```
register1 = count
register1 = register1 + 1
count = register1
```

where `register1` is one of the local CPU registers. Similarly, the statement “`count--`” is implemented as follows:

```
register2 = count
register2 = register2 - 1
count = register2
```

where again `register2` is one of the local CPU registers. Even though `register1` and `register2` may be the same physical register, remember that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

The concurrent execution of “`count++`” and “`count--`” is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

T_0 :	<i>producer</i>	execute	$register_1 = count$	{ $register_1 = 5$ }
T_1 :	<i>producer</i>	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
T_2 :	<i>consumer</i>	execute	$register_2 = count$	{ $register_2 = 5$ }
T_3 :	<i>consumer</i>	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
T_4 :	<i>producer</i>	execute	$count = register_1$	{ $count = 6$ }
T_5 :	<i>consumer</i>	execute	$count = register_2$	{ $count = 4$ }

Notice that we have arrived at the incorrect state “`count == 4`”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5 , we would arrive at the incorrect state “`count == 6`”.

We would arrive at this incorrect state because we allowed both processes to manipulate the variable `count` concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable `count`. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the prominence of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads—which are quite possibly sharing data—are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one

another. Because of the importance of this issue, we devote a major portion of this chapter to **process synchronization** and **coordination** among cooperating processes.

6.2 The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so-called critical-section problem. Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The ***critical-section problem*** is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion.** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

```

while (true) {
    entry section
    critical section
    exit section
    remainder section
}

```

Figure 6.1 General structure of a typical process.

- 3. Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (*kernel code*) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Another example is illustrated in Figure 6.2. In this situation, two processes, P_0 and P_1 , are creating child processes using the `fork()` system call. Recall from Section 3.3.1 that `fork()` returns the process identifier of the newly created process to the parent process. In this example, there is a race condition on the variable kernel variable `next_available_pid` which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence

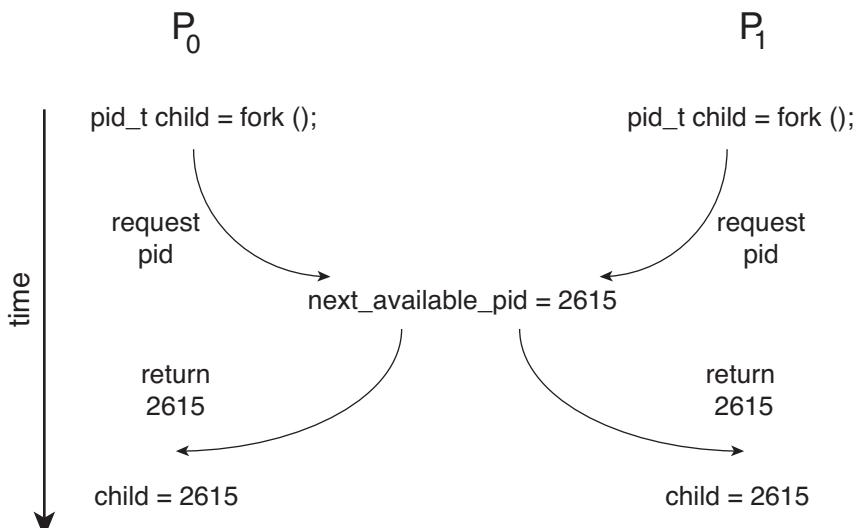


Figure 6.2 Race condition when assigning a pid.

of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different CPU cores.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. (Of course, this risk can also be minimized by designing kernel code that does not behave in this way.) Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

6.3 Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1 . For convenience, when presenting P_i , we use P_j to denote the other process; that is, j equals $1 - i$.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

```

        while (true) {
            flag[i] = true;
            turn = j;
            while (flag[j] && turn == j)
                ;
            /* critical section */

            flag[i] = false;

            /*remainder section */
        }
    
```

Figure 6.3 The structure of process P_i in Peterson's solution.

The variable `turn` indicates whose turn it is to enter its critical section. That is, if `turn == i`, then process P_i is allowed to execute in its critical section. The `flag` array is used to indicate if a process is ready to enter its critical section. For example, if `flag[i]` is `true`, P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 6.3.

To enter the critical section, process P_i first sets `flag[i]` to be `true` and then sets `turn` to the value j , thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, `turn` will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of `turn` determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either `flag[j] == false` or `turn == i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0] == flag[1] == true`. These two observations imply that P_0 and P_1 could not have successfully executed their `while` statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes—say, P_j —must have successfully executed the `while` statement, whereas P_i had to execute at least one additional statement (“`turn == j`”). However, at that time, `flag[j] == true` and `turn == j`, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition $\text{flag}[j] == \text{true}$ and $\text{turn} == j$; this loop is the only one possible. If P_j is not ready to enter the critical section, then $\text{flag}[j] == \text{false}$, and P_i can enter its critical section. If P_j has set $\text{flag}[j]$ to true and is also executing in its while statement, then either $\text{turn} == i$ or $\text{turn} == j$. If $\text{turn} == i$, then P_i will enter the critical section. If $\text{turn} == j$, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset $\text{flag}[j]$ to false, allowing P_i to enter its critical section. If P_j resets $\text{flag}[j]$ to true, it must also set turn to i . Thus, since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

As mentioned at the beginning of this section, Peterson's solution is not guaranteed to work on modern computer architectures for the primary reason that, to improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies. For a single-threaded application, this reordering is immaterial as far as program correctness is concerned, as the final values are consistent with what is expected. (This is similar to balancing a checkbook—the actual order in which credit and debit operations are performed is unimportant, because the final balance will still be the same.) But for a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.

As an example, consider the following data that are shared between two threads:

```
boolean flag = false;
int x = 0;
```

where Thread 1 performs the statements

```
while (!flag)
    ;
print x;
```

and Thread 2 performs

```
x = 100;
flag = true;
```

The expected behavior is, of course, that Thread 1 outputs the value 100 for variable x . However, as there are no data dependencies between the variables flag and x , it is possible that a processor may reorder the instructions for Thread 2 so that flag is assigned true before assignment of $x = 100$. In this situation, it is possible that Thread 1 would output 0 for variable x . Less obvious is that the processor may also reorder the statements issued by Thread 1 and load the variable x before loading the value of flag . If this were to occur, Thread 1 would output 0 for variable x even if the instructions issued by Thread 2 were not reordered.

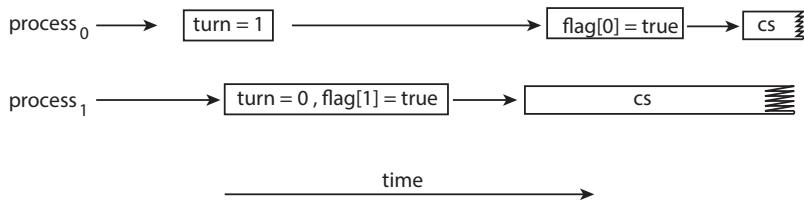


Figure 6.4 The effects of instruction reordering in Peterson’s solution.

How does this affect Peterson’s solution? Consider what happens if the assignments of the first two statements that appear in the entry section of Peterson’s solution in Figure 6.3 are reordered; it is possible that both threads may be active in their critical sections at the same time, as shown in Figure 6.4.

As you will see in the following sections, the only way to preserve mutual exclusion is by using proper synchronization tools. Our discussion of these tools begins with primitive support in hardware and proceeds through abstract, high-level, software-based APIs available to both kernel developers and application programmers.

6.4 Hardware Support for Synchronization

We have just described one software-based solution to the critical-section problem. (We refer to it as a *software-based* solution because the algorithm involves no special support from the operating system or specific hardware instructions to ensure mutual exclusion.) However, as discussed, software-based solutions are not guaranteed to work on modern computer architectures. In this section, we present three hardware instructions that provide support for solving the critical-section problem. These primitive operations can be used directly as synchronization tools, or they can be used to form the foundation of more abstract synchronization mechanisms.

6.4.1 Memory Barriers

In Section 6.3, we saw that a system may reorder instructions, a policy that can lead to unreliable data states. How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**. In general, a memory model falls into one of two categories:

1. **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.
2. **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.

Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor. To address this issue, computer architectures provide instructions that can *force* any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to

threads running on other processors. Such instructions are known as **memory barriers** or **memory fences**. When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed. Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Let's return to our most recent example, in which reordering of instructions could have resulted in the wrong output, and use a memory barrier to ensure that we obtain the expected output.

If we add a memory barrier operation to Thread 1

```
while (!flag)
    memory_barrier();
print x;
```

we guarantee that the value of `flag` is loaded before the value of `x`.

Similarly, if we place a memory barrier between the assignments performed by Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

we ensure that the assignment to `x` occurs before the assignment to `flag`.

With respect to Peterson's solution, we could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations shown in Figure 6.4. Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

6.4.2 Hardware Instructions

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the `test_and_set()` and `compare_and_swap()` instructions.

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

Figure 6.5 The definition of the atomic `test_and_set()` instruction.

```

do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);

```

Figure 6.6 Mutual-exclusion implementation with `test_and_set()`.

The `test_and_set()` instruction can be defined as shown in Figure 6.5. The important characteristic of this instruction is that it is executed atomically. Thus, if two `test_and_set()` instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order. If the machine supports the `test_and_set()` instruction, then we can implement mutual exclusion by declaring a boolean variable `lock`, initialized to `false`. The structure of process P_i is shown in Figure 6.6.

The `compare_and_swap()` instruction (CAS), just like the `test_and_set()` instruction, operates on two words atomically, but uses a different mechanism that is based on swapping the content of two words.

The CAS instruction operates on three operands and is defined in Figure 6.7. The operand `value` is set to `new_value` only if the expression `(*value == expected)` is true. Regardless, CAS always returns the original value of the variable `value`. The important characteristic of this instruction is that it is executed atomically. Thus, if two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

Mutual exclusion using CAS can be provided as follows: A global variable (`lock`) is declared and is initialized to 0. The first process that invokes `compare_and_swap()` will set `lock` to 1. It will then enter its critical section,

```

int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}

```

Figure 6.7 The definition of the atomic `compare_and_swap()` instruction.

```

while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}

```

Figure 6.8 Mutual exclusion with the compare_and_swap() instruction.

because the original value of `lock` was equal to the expected value of 0. Subsequent calls to `compare_and_swap()` will not succeed, because `lock` now is not equal to the expected value of 0. When a process exits its critical section, it sets `lock` back to 0, which allows another process to enter its critical section. The structure of process P_i is shown in Figure 6.8.

Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement. In Figure 6.9, we present

```

while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = 0;
    else
        waiting[j] = false;

    /* remainder section */
}

```

Figure 6.9 Bounded-waiting mutual exclusion with compare_and_swap().

MAKING COMPARE-AND-SWAP ATOMIC

On Intel x86 architectures, the assembly language statement `cpxchq` is used to implement the `compare_and_swap()` instruction. To enforce atomic execution, the `lock` prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cpxchq <destination operand>, <source operand>
```

another algorithm using the `compare_and_swap()` instruction that satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n] ;
int lock;
```

The elements in the `waiting` array are initialized to `false`, and `lock` is initialized to 0. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section only if either `waiting[i] == false` or `key == 0`. The value of `key` can become 0 only if the `compare_and_swap()` is executed. The first process to execute the `compare_and_swap()` will find `key == 0`; all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, maintaining the mutual-exclusion requirement.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to 0 or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array `waiting` in the cyclic ordering $(i + 1, i + 2, \dots, n - 1, 0, \dots, i - 1)$. It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Details describing the implementation of the `atomic test_and_set()` and `compare_and_swap()` instructions are discussed more fully in books on computer architecture.

6.4.3 Atomic Variables

Typically, the `compare_and_swap()` instruction is not used directly to provide mutual exclusion. Rather, it is used as a basic building block for constructing other tools that solve the critical-section problem. One such tool is an **atomic variable**, which provides atomic operations on basic data types such as integers and booleans. We know from Section 6.1 that incrementing or decrementing an integer value may produce a race condition. Atomic variables can be used in to ensure mutual exclusion in situations where there may be a data race on a single variable while it is being updated, as when a counter is incremented.

Most systems that support atomic variables provide special atomic data types as well as functions for accessing and manipulating atomic variables.

These functions are often implemented using `compare_and_swap()` operations. As an example, the following increments the atomic integer sequence:

```
increment(&sequence);
```

where the `increment()` function is implemented using the CAS instruction:

```
void increment(atomic_int *v)
{
    int temp;

    do {
        temp = *v;
    }
    while (temp != compare_and_swap(v, temp, temp+1));
}
```

It is important to note that although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstances. For example, in the bounded-buffer problem described in Section 6.1, we could use an atomic integer for `count`. This would ensure that the updates to `count` were atomic. However, the producer and consumer processes also have `while` loops whose condition depends on the value of `count`. Consider a situation in which the buffer is currently empty and two consumers are looping while waiting for $\text{count} > 0$. If a producer entered one item in the buffer, both consumers could exit their `while` loops (as `count` would no longer be equal to 0) and proceed to consume, even though the value of `count` was only set to 1.

Atomic variables are commonly used in operating systems as well as concurrent applications, although their use is often limited to single updates of shared data such as counters and sequence generators. In the following sections, we explore more robust tools that address race conditions in more generalized situations.

6.5 Mutex Locks

The hardware-based solutions to the critical-section problem presented in Section 6.4 are complicated as well as generally inaccessible to application programmers. Instead, operating-system designers build higher-level software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. (In fact, the term *mutex* is short for *mutual exclusion*.) We use the mutex lock to protect critical sections and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The `acquire()` function acquires the lock, and the `release()` function releases the lock, as illustrated in Figure 6.10.

A mutex lock has a boolean variable `available` whose value indicates if the lock is available or not. If the lock is available, a call to `acquire()` succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

```

while (true) {
    acquire lock
    critical section
    release lock
    remainder section
}

```

Figure 6.10 Solution to the critical-section problem using mutex locks.

The definition of `acquire()` is as follows:

```

acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}

```

The definition of `release()` is as follows:

```

release() {
    available = true;
}

```

Calls to either `acquire()` or `release()` must be performed atomically. Thus, mutex locks can be implemented using the CAS operation described in Section 6.4, and we leave the description of this technique as an exercise.

LOCK CONTENTION

Locks are either contended or uncontended. A lock is considered **contended** if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**. Contended locks can experience either *high contention* (a relatively large number of threads attempting to acquire the lock) or *low contention* (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.

WHAT IS MEANT BY “SHORT DURATION”?

Spinlocks are often identified as the locking mechanism of choice on multiprocessor systems when the lock is to be held for a short duration. But what exactly constitutes a *short duration*? Given that waiting on a lock requires two context switches—a context switch to move the thread to the waiting state and a second context switch to restore the waiting thread once the lock becomes available—the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively. (In Section 6.6, we examine a strategy that avoids busy waiting by temporarily putting the waiting process to sleep and then awakening it once the lock becomes available.)

The type of mutex lock we have been describing is also called a **spin-lock** because the process “spins” while waiting for the lock to become available. (We see the same issue with the code examples illustrating the `compare_and_swap()` instruction.) Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking. If a lock is to be held for a short duration, one thread can “spin” on one processing core while another thread performs its critical section on another core. On modern multicore computing systems, spinlocks are widely used in many operating systems.

In Chapter 7 we examine how mutex locks can be used to solve classical synchronization problems. We also discuss how mutex locks and spinlocks are used in several operating systems, as well as in Pthreads.

6.6 Semaphores

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

A **semaphore** `S` is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the `wait()` operation was originally termed `P` (from the Dutch

proberen, “to test”); `signal()` was originally called `V` (from *verhogen*, “to increment”). The definition of `wait()` is as follows:

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

The definition of `signal()` is as follows:

```
signal(S) {
    S++;
}
```

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed atomically. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of `S` ($S \leq 0$), as well as its possible modification (`S--`), must be executed without interruption. We shall see how these operations can be implemented in Section 6.6.2. First, let’s see how semaphores can be used.

6.6.1 Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2 . Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore `synch`, initialized to 0. In process P_1 , we insert the statements

```
 $S_1;$ 
signal(synch);
```

In process P_2 , we insert the statements

```
wait(synch);
S2;
```

Because `synch` is initialized to 0, P_2 will execute S_2 only after P_1 has invoked `signal(synch)`, which is after statement S_1 has been executed.

6.6.2 Semaphore Implementation

Recall that the implementation of mutex locks discussed in Section 6.5 suffers from busy waiting. The definitions of the `wait()` and `signal()` semaphore operations just described present the same problem. To overcome this problem, we can modify the definition of the `wait()` and `signal()` operations as follows: When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is suspended, waiting on a semaphore S , should be restarted when some other process executes a `signal()` operation. The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer `value` and a list of processes `list`. When a process must wait on a semaphore, it is added to the list of processes. A `signal()` operation removes one process from the list of waiting processes and awakens that process.

Now, the `wait()` semaphore operation can be defined as

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}
```

and the `signal()` semaphore operation can be defined as

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The `sleep()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a suspended process *P*. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the `wait()` operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use any queuing strategy. Correct usage of semaphores does not depend on a particular queuing strategy for the semaphore lists.

As mentioned, it is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time. This is a critical-section problem, and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the `wait()` and `signal()` operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multicore environment, interrupts must be disabled on every processing core. Otherwise, instructions from different processes (running on different cores) may be interleaved in some arbitrary way. Disabling interrupts on every core can be a difficult task and can seriously diminish performance. Therefore, SMP systems must provide alternative techniques—such as `compare_and_swap()` or spinlocks—to ensure that `wait()` and `signal()` are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait()` and `signal()` operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `wait()` and `signal()` operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs

rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

6.7 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place, and these sequences do not always occur.

We have seen an example of such errors in the use of a count in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the count value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that mutex locks and semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when either mutex locks or semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a binary semaphore variable `mutex`, which is initialized to 1. Each process must execute `wait(mutex)` before entering the critical section and `signal(mutex)` afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we list several difficulties that may result. Note that these difficulties will arise even if a *single* process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a program interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution:

```
signal(mutex);
...
critical section
...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a program replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

```
wait(mutex);
...
critical section
...
wait(mutex);
```

In this case, the process will permanently block on the second call to `wait()`, as the semaphore is now unavailable.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or the process will permanently block.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem. One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

6.7.1 Monitor Usage

An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A **monitor type** is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an

```

monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . .
    }

    function P2 ( . . . ) {
        . .
    }

    .

    .

    function Pn ( . . . ) {
        . .
    }

    initialization_code ( . . . ) {
        . .
    }
}
```

Figure 6.11 Pseudocode syntax of a monitor.

instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is shown in Figure 6.11. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.12). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

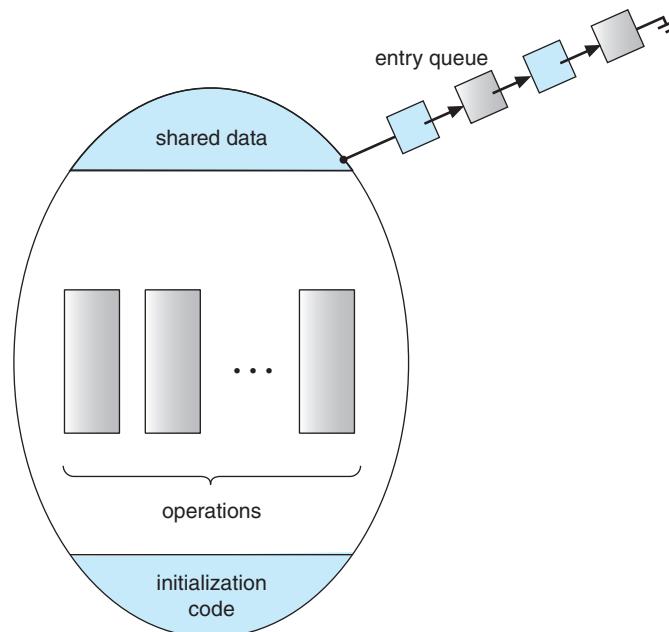


Figure 6.12 Schematic view of a monitor.

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal()` operation has no effect; that is, the state of `x` is the same as if the operation had never been executed (Figure 6.13). Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore.

Now suppose that, when the `x.signal()` operation is invoked by a process `P`, there exists a suspended process `Q` associated with condition `x`. Clearly, if the suspended process `Q` is allowed to resume its execution, the signaling process `P` must wait. Otherwise, both `P` and `Q` would be active simultaneously within the monitor. Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

1. **Signal and wait.** `P` either waits until `Q` leaves the monitor or waits for another condition.
2. **Signal and continue.** `Q` either waits until `P` leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since `P` was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other, if we allow thread `P` to continue, then by the time `Q` is resumed, the logical condition for which `Q` was waiting may no longer hold. A compromise between these two choices exists as well: when thread `P` executes the `signal` operation, it immediately leaves the monitor. Hence, `Q` is immediately resumed.

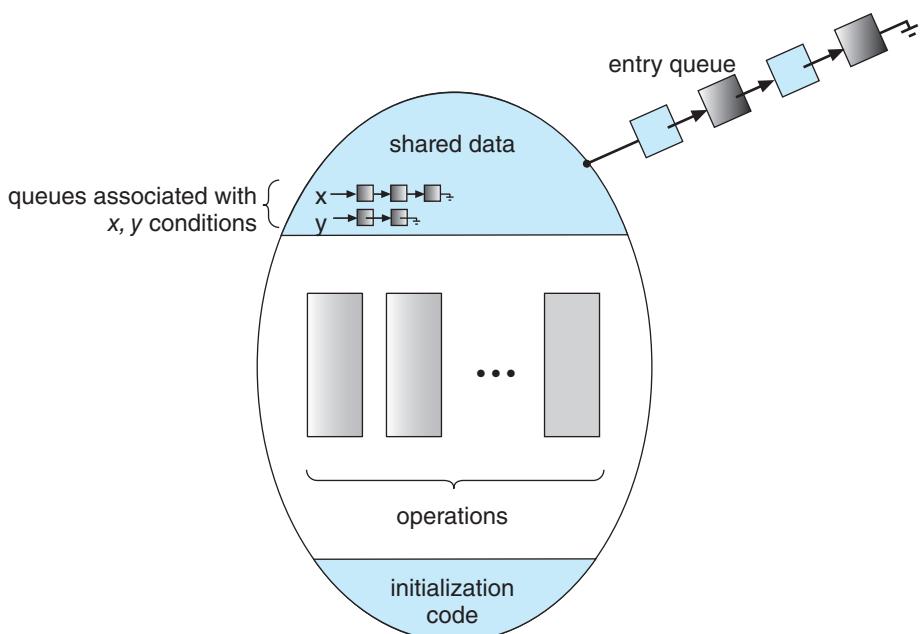


Figure 6.13 Monitor with condition variables.

Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C#. Other languages—such as Erlang—provide concurrency support using a similar mechanism.

6.7.2 Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a binary semaphore `mutex` (initialized to 1) is provided to ensure mutual exclusion. A process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor.

We will use the signal-and-wait scheme in our implementation. Since a signaling process must wait until the resumed process either leaves or waits, an additional binary semaphore, `next`, is introduced, initialized to 0. The signaling processes can use `next` to suspend themselves. An integer variable `next_count` is also provided to count the number of processes suspended on `next`. Thus, each external function `F` is replaced by

```
wait(mutex);
...
body of F
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented as well. For each condition `x`, we introduce a binary semaphore `x_sem` and an integer variable `x_count`, both initialized to 0. The operation `x.wait()` can now be implemented as

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

The operation `x.signal()` can be implemented as

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen (see the bibliographical notes at the end of the chapter). In some cases, however, the generality of the implementation is

```

monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }

    void release() {
        busy = false;
        x.signal();
    }

    initialization_code() {
        busy = false;
    }
}

```

Figure 6.14 A monitor to allocate a single resource.

unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.27.

6.7.3 Resuming Processes within a Monitor

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition *x*, and an *x.signal()* operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. In these circumstances, the **conditional-wait** construct can be used. This construct has the form

x.wait(c);

where *c* is an integer expression that is evaluated when the *wait()* operation is executed. The value of *c*, which is called a **priority number**, is then stored with the name of the process that is suspended. When *x.signal()* is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the *ResourceAllocator* monitor shown in Figure 6.14, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation

request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);  
...  
access the resource;  
...  
R.release();
```

where R is an instance of type `ResourceAllocator`.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the `ResourceAllocator` monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the `ResourceAllocator` monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only through the use of the additional mechanisms that are described in Chapter 17.

6.8 Liveness

One consequence of using synchronization tools to coordinate access to critical sections is the possibility that a process attempting to enter its critical section will wait indefinitely. Recall that in Section 6.2, we outlined three criteria that solutions to the critical-section problem must satisfy. Indefinite waiting violates two of these—the progress and bounded-waiting criteria.

Liveness refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle. A process waiting indefinitely under the circumstances just described is an example of a “liveness failure.”

There are many different forms of liveness failure; however, all are generally characterized by poor performance and responsiveness. A very simple example of a liveness failure is an infinite loop. A busy wait loop presents the *possibility* of a liveness failure, especially if a process may loop an arbitrarily long period of time. Efforts at providing mutual exclusion using tools such as mutex locks and semaphores can often lead to such failures in concurrent programming. In this section, we explore two situations that can lead to liveness failures.

6.8.1 Deadlock

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, P_0 and P_1 , each accessing two semaphores, S and Q, set to the value 1:

P_0	P_1
<code>wait(S);</code> <code>wait(Q);</code> <code>.</code> <code>.</code> <code>signal(S);</code> <code>signal(Q);</code>	<code>wait(Q);</code> <code>wait(S);</code> <code>.</code> <code>.</code> <code>signal(Q);</code> <code>signal(S);</code>

Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlocked state when *every process in the set is waiting for an event that can be caused only by another process in the set*. The “events” with which we are mainly concerned here are the acquisition and release of resources such as mutex locks and semaphores. Other types of events may result in deadlocks, as we show in more detail in Chapter 8. In

that chapter, we describe various mechanisms for dealing with the deadlock problem, as well as other forms of liveness failures.

6.8.2 Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

As an example, assume we have three processes— L , M , and H —whose priorities follow the order $L < M < H$. Assume that process H requires a semaphore S , which is currently being accessed by process L . Ordinarily, process H would wait for L to finish using resource S . However, now suppose that process M becomes runnable, thereby preempting process L . Indirectly, a process with a lower priority—process M —has affected how long process H must wait for L to relinquish resource S .

This liveness problem is known as **priority inversion**, and it can occur only in systems with more than two priorities. Typically, priority inversion is avoided by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H , thereby preventing process M from preempting its execution. When process L had finished using resource S , it would relinquish its inherited priority from H and assume its original priority. Because resource S would now be available, process H —not M —would run next.

6.9 Evaluation

We have described several different synchronization tools that can be used to solve the critical-section problem. Given correct implementation and usage, these tools can be used effectively to ensure mutual exclusion as well as address liveness issues. With the growth of concurrent programs that leverage the power of modern multicore computer systems, increasing attention is being paid to the performance of synchronization tools. Trying to identify when to use which tool, however, can be a daunting challenge. In this section, we present some simple strategies for determining when to use specific synchronization tools.

The hardware solutions outlined in Section 6.4 are considered very low level and are typically used as the foundations for constructing other synchronization tools, such as mutex locks. However, there has been a recent focus on using the CAS instruction to construct **lock-free** algorithms that provide protection from race conditions without requiring the overhead of locking. Although these lock-free solutions are gaining popularity due to low overhead

PRIORITY INVERSION AND THE MARS PATHFINDER

Priority inversion can be more than a scheduling inconvenience. On systems with tight time constraints—such as real-time systems—priority inversion can cause a process to take longer than it should to accomplish a task. When that happens, other failures can cascade, resulting in system failure.

Consider the Mars Pathfinder, a NASA space probe that landed a robot, the Sojourner rover, on Mars in 1997 to conduct experiments. Shortly after the Sojourner began operating, it started to experience frequent computer resets. Each reset reinitialized all hardware and software, including communications. If the problem had not been solved, the Sojourner would have failed in its mission.

The problem was caused by the fact that one high-priority task, “bc_dist,” was taking longer than expected to complete its work. This task was being forced to wait for a shared resource that was held by the lower-priority “ASI/MET” task, which in turn was preempted by multiple medium-priority tasks. The “bc_dist” task would stall waiting for the shared resource, and ultimately the “bc_sched” task would discover the problem and perform the reset. The Sojourner was suffering from a typical case of priority inversion.

The operating system on the Sojourner was the VxWorks real-time operating system, which had a global variable to enable priority inheritance on all semaphores. After testing, the variable was set on the Sojourner (on Mars!), and the problem was solved.

A full description of the problem, its detection, and its solution was written by the software team lead and is available at http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html.

and ability to scale, the algorithms themselves are often difficult to develop and test. (In the exercises at the end of this chapter, we ask you to evaluate the correctness of a lock-free stack.)

CAS-based approaches are considered an *optimistic* approach—you optimistically first update a variable and then use collision detection to see if another thread is updating the variable concurrently. If so, you repeatedly retry the operation until it is successfully updated without conflict. Mutual-exclusion locking, in contrast, is considered a *pessimistic* strategy; you assume another thread is concurrently updating the variable, so you pessimistically acquire the lock before making any updates.

The following guidelines identify general rules concerning performance differences between CAS-based synchronization and traditional synchronization (such as mutex locks and semaphores) under varying contention loads:

- **Uncontented.** Although both options are generally fast, CAS protection will be somewhat faster than traditional synchronization.
- **Moderate contention.** CAS protection will be faster—possibly much faster—than traditional synchronization.

- **High contention.** Under very highly contended loads, traditional synchronization will ultimately be faster than CAS-based synchronization.

Moderate contention is particularly interesting to examine. In this scenario, the CAS operation succeeds most of the time, and when it fails, it will iterate through the loop shown in Figure 6.8 only a few times before ultimately succeeding. By comparison, with mutual-exclusion locking, *any* attempt to acquire a contended lock will result in a more complicated—and time-intensive—code path that suspends a thread and places it on a wait queue, requiring a context switch to another thread.

The choice of a mechanism that addresses race conditions can also greatly affect system performance. For example, atomic integers are much lighter weight than traditional locks, and are generally more appropriate than mutex locks or semaphores for single updates to shared variables such as counters. We also see this in the design of operating systems where spinlocks are used on multiprocessor systems when locks are held for short durations. In general, mutex locks are simpler and require less overhead than semaphores and are preferable to binary semaphores for protecting access to a critical section. However, for some uses—such as controlling access to a finite number of resources—a counting semaphore is generally more appropriate than a mutex lock. Similarly, in some instances, a reader–writer lock may be preferred over a mutex lock, as it allows a higher degree of concurrency (that is, multiple readers).

The appeal of higher-level tools such as monitors and condition variables is based on their simplicity and ease of use. However, such tools may have significant overhead and, depending on their implementation, may be less likely to scale in highly contended situations.

Fortunately, there is much ongoing research toward developing scalable, efficient tools that address the demands of concurrent programming. Some examples include:

- Designing compilers that generate more efficient code.
- Developing languages that provide support for concurrent programming.
- Improving the performance of existing libraries and APIs.

In the next chapter, we examine how various operating systems and APIs available to developers implement the synchronization tools presented in this chapter.

6.10 Summary

- A race condition occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.
- A critical section is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem

is to design a protocol whereby processes can synchronize their activity to cooperatively share data.

- A solution to the critical-section problem must satisfy the following three requirements: (1) mutual exclusion, (2) progress, and (3) bounded waiting. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.
- Software solutions to the critical-section problem, such as Peterson's solution, do not work well on modern computer architectures.
- Hardware support for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.
- A mutex lock provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.
- Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.
- A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
- Solutions to the critical-section problem may suffer from liveness problems, including deadlock.
- The various tools that can be used to solve the critical-section problem as well as to synchronize the activity of processes can be evaluated under varying levels of contention. Some tools work better under certain contention loads than others.

Practice Exercises

- 6.1 In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.
- 6.2 What is the meaning of the term *busy waiting*? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.
- 6.3 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.
- 6.4 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

- 6.5 Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.
- 6.6 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Further Reading

The mutual-exclusion problem was first discussed in a classic paper by [Dijkstra (1965)]. The semaphore concept was suggested by [Dijkstra (1965)]. The monitor concept was developed by [Brinch-Hansen (1973)]. [Hoare (1974)] gave a complete description of the monitor.

For more on the Mars Pathfinder problem see http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

A thorough discussion of memory barriers and cache memory is presented in [Mckenney (2010)]. [Herlihy and Shavit (2012)] presents details on several issues related to multiprocessor programming, including memory models and compare-and-swap instructions. [Babra (2013)] examines nonblocking algorithms on modern multicore systems.

Bibliography

- [Babra (2013)] S. A. Babra, “Nonblocking Algorithms and Scalable Multicore Programming”, *ACM queue*, Volume 11, Number 5 (2013).
- [Brinch-Hansen (1973)] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).
- [Dijkstra (1965)] E. W. Dijkstra, “Cooperating Sequential Processes”, Technical report, Technological University, Eindhoven, the Netherlands (1965).
- [Herlihy and Shavit (2012)] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).
- [Hoare (1974)] C. A. R. Hoare, “Monitors: An Operating System Structuring Concept”, *Communications of the ACM*, Volume 17, Number 10 (1974), pages 549–557.
- [Mckenney (2010)] P. E. Mckenney, “Memory Barriers: a Hardware View for Software Hackers” (2010).

Chapter 6 Exercises

- 6.7 The pseudocode of Figure 6.15 illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:
- What data have a race condition?
 - How could the race condition be fixed?
- 6.8 Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
    if (amount > highestBid)
        highestBid = amount;
}
```

```
push(item) {
    if (top < SIZE) {
        stack[top] = item;
        top++;
    }
    else
        ERROR
}

pop() {
    if (!is_empty()) {
        top--;
        return stack[top];
    }
    else
        ERROR
}

is_empty() {
    if (top == 0)
        return true;
    else
        return false;
}
```

Figure 6.16 Array-based stack for Exercise 6.12.

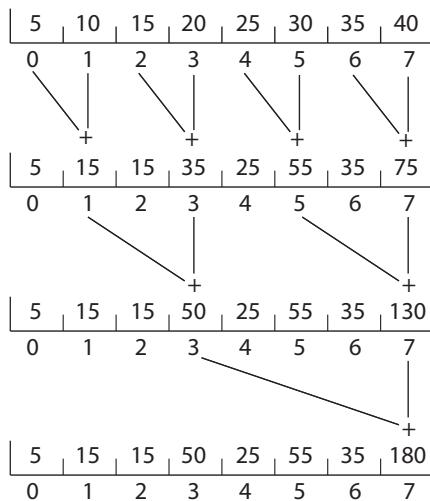


Figure 6.17 Summing an array as a series of partial sums for Exercise 6.14.

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

- 6.9 The following program example can be used to sum the array `values` of size N elements in parallel on a system containing N computing cores (there is a separate processor for each array element):

```
for j = 1 to log_2(N) {
    for k = 1 to N {
        if ((k + 1) % pow(2,j) == 0) {
            values[k] += values[k - pow(2,(j-1))]
        }
    }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in Figure 6.16. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.

- 6.10 The `compare_and_swap()` instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in Figure 6.17 presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of `Node` elements with `top` representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {
    value_t data;
    struct node *next;
} Node;

Node *top; // top of stack

void push(value_t item) {
    Node *old_node;
    Node *new_node;

    new_node = malloc(sizeof(Node));
    new_node->data = item;

    do {
        old_node = top;
        new_node->next = old_node;
    }
    while (compare_and_swap(top, old_node, new_node) != old_node);
}

value_t pop() {
    Node *old_node;
    Node *new_node;

    do {
        old_node = top;
        if (old_node == NULL)
            return NULL;
        new_node = old_node->next;
    }
    while (compare_and_swap(top, old_node, new_node) != old_node);

    return old_node->data;
}
```

Figure 6.18 Lock-free stack for Exercise 6.15.

- 6.11 One approach for using `compare_and_swap()` for implementing a spin-lock is as follows:

```
void lock_spinlock(int *lock) {
    while (compare_and_swap(lock, 0, 1) != 0)
        ; /* spin */
}
```

A suggested alternative approach is to use the “compare and compare-and-swap” idiom, which checks the status of the lock before invoking the

`compare_and_swap()` operation. (The rationale behind this approach is to invoke `compare_and_swap()` only if the lock is currently available.) This strategy is shown below:

```
void lock_spinlock(int *lock) {
{
    while (true) {
        if (*lock == 0) {
            /* lock appears to be available */

            if (!compare_and_swap(lock, 0, 1))
                break;
        }
    }
}
```

Does this “compare and compare-and-swap” idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

- 6.12 Some semaphore implementations provide a function `getValue()` that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling `wait()` so that a process will only call `wait()` if the value of the semaphore is > 0 , thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
    wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function `getValue()` in this scenario.

- 6.13 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 , share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i ($i == 0$ or 1) is shown in Figure 6.18. The other process is \bar{P}_j ($j == 1$ or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

- 6.14 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

```
while (true) {
    flag[i] = true;

    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; /* do nothing */
            flag[i] = true;
        }
    }

    /* critical section */

    turn = j;
    flag[i] = false;

    /* remainder section */
}
```

Figure 6.19 The structure of process P_i in Dekker's algorithm.

All the elements of `flag` are initially `idle`. The initial value of `turn` is immaterial (between 0 and $n-1$). The structure of process P_i is shown in Figure 6.19. Prove that the algorithm satisfies all three requirements for the critical-section problem.

- 6.15 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.
- 6.16 Consider how to implement a mutex lock using the `compare_and_swap()` instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
    int available;
} lock;
```

The value (`available == 0`) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this `struct`, illustrate how the following functions can be implemented using the `compare_and_swap()` instruction:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

```

while (true) {
    while (true) {
        flag[i] = want_in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            } else
                j = (j + 1) % n;
        }

        flag[i] = in_cs;
        j = 0;

        while ((j < n) && (j == i || flag[j] != in_cs))
            j++;

        if ((j >= n) && (turn == i || flag[turn] == idle))
            break;
    }

    /* critical section */

    j = (turn + 1) % n;

    while (flag[j] == idle)
        j = (j + 1) % n;

    turn = j;
    flag[i] = idle;

    /* remainder section */
}

```

Figure 6.20 The structure of process P_i in Eisenberg and McGuire's algorithm.

- 6.17 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
- 6.18 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.
- 6.19 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a

spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

- 6.20** Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.
- 6.21** A multithreaded web server wishes to keep track of the number of requests it services (known as *hits*). Consider the two following strategies to prevent a race condition on the variable *hits*. The first strategy is to use a basic mutex lock when updating *hits*:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

- 6.22** Consider the code example for allocating and releasing processes shown in Figure 6.20.
- a. Identify the race condition(s).
 - b. Assume you have a mutex lock named *mutex* with the operations *acquire()* and *release()*. Indicate where the locking needs to be placed to prevent the race condition(s).
 - c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0
```

to prevent the race condition(s)?

- 6.23** Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is

```

#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
    int new_pid;

    if (number_of_processes == MAX_PROCESSES)
        return -1;
    else {
        /* allocate necessary process resources */
        ++number_of_processes;

        return new_pid;
    }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}

```

Figure 6.21 Allocating and releasing processes for Exercise 6.27.

released. Illustrate how semaphores can be used by a server to limit the number of concurrent connections.

- 6.24** In Section 6.7, we use the following illustration as an incorrect use of semaphores to solve the critical-section problem:

```

wait(mutex);
...
critical section
...
wait(mutex);

```

Explain why this is an example of a liveness failure.

- 6.25** Demonstrate that monitors and semaphores are equivalent to the degree that they can be used to implement solutions to the same types of synchronization problems.
- 6.26** Describe how the `signal()` operation associated with monitors differs from the corresponding operation defined for semaphores.
- 6.27** Suppose the `signal()` statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 6.7 can be simplified in this situation.

- 6.28** Consider a system consisting of processes P_1, P_2, \dots, P_n , each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.
- 6.29** A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than n . Write a monitor to coordinate access to the file.
- 6.30** When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?
- 6.31** Design an algorithm for a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.
- 6.32** Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

Programming Problems

- 6.33 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of *licenses*, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such a request will be granted only when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the `decrease_count()` function:

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
    if (available_resources < count)
        return -1;
    else {
        available_resources -= count;

        return 0;
    }
}
```

When a process wants to return a number of resources, it calls the `increase_count()` function:

```
/* increase available_resources by count */
int increase_count(int count) {
    available_resources += count;

    return 0;
}
```

The preceding program segment produces a race condition. Do the following:

- a. Identify the data involved in the race condition.

- b. Identify the location (or locations) in the code where the race condition occurs.
 - c. Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the `decrease_count()` function so that the calling process is blocked until sufficient resources are available.
- 6.34** The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and -1 otherwise. This leads to awkward programming for a process that wishes to obtain a number of resources:

```
while (decrease_count(count) == -1)  
    ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

The process will return from this function call only when sufficient resources are available.

Synchronization Examples



In Chapter 6, we presented the critical-section problem and focused on how race conditions can occur when multiple concurrent processes share data. We went on to examine several tools that address the critical-section problem by preventing race conditions from occurring. These tools ranged from low-level hardware solutions (such as memory barriers and the compare-and-swap operation) to increasingly higher-level tools (from mutex locks to semaphores to monitors). We also discussed various challenges in designing applications that are free from race conditions, including liveness hazards such as deadlocks. In this chapter, we apply the tools presented in Chapter 6 to several classic synchronization problems. We also explore the synchronization mechanisms used by the Linux, UNIX, and Windows operating systems, and we describe API details for both Java and POSIX systems.

CHAPTER OBJECTIVES

- Explain the bounded-buffer, readers-writers, and dining-philosophers synchronization problems.
- Describe specific tools used by Linux and Windows to solve process synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems.
- Design and develop solutions to process synchronization problems using POSIX and Java APIs.

7.1 Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the

```
while (true) {
    . .
    /* produce an item in next_produced */
    . .
    wait(empty);
    wait(mutex);
    . .
    /* add next_produced to the buffer */
    . .
    signal(mutex);
    signal(full);
}
```

Figure 7.1 The structure of the producer process.

traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

7.1.1 The Bounded-Buffer Problem

The *bounded-buffer problem* was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The `mutex` binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers. The semaphore `empty` is initialized to the value n ; the semaphore `full` is initialized to the value 0.

The code for the producer process is shown in Figure 7.1, and the code for the consumer process is shown in Figure 7.2. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

7.1.2 The Readers-Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, read and write) the database. We distinguish

```

        while (true) {
            wait(full);
            wait(mutex);
            . . .
            /* remove an item from buffer to next_consumed */
            . . .
            signal(mutex);
            signal(empty);
            . . .
            /* consume the item in next_consumed */
            . . .
        }
    
```

Figure 7.2 The structure of the consumer process.

between these two types of processes by referring to the former as *readers* and to the latter as *writers*. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers-writers problem**. Since it was originally stated, it has been used to test nearly every new synchronization primitive.

The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers-writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers-writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers-writers problem. See the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers-writers problem.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

```

semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;

```

The binary semaphores `mutex` and `rw_mutex` are initialized to 1; `read_count` is a counting semaphore initialized to 0. The semaphore `rw_mutex`

```

        while (true) {
            wait(rw_mutex);
            . . .
            /* writing is performed */
            . . .
            signal(rw_mutex);
        }
    
```

Figure 7.3 The structure of a writer process.

is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `read_count` is updated. The `read_count` variable keeps track of how many processes are currently reading the object. The semaphore `rw_mutex` functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers that enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 7.3; the code for a reader process is shown in Figure 7.4. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `rw_mutex`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(rw_mutex)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers-writers problem and its solutions have been generalized to provide **reader-writer** locks on some systems. Acquiring a reader-writer lock requires specifying the mode of the lock: either *read* or *write* access. When a

```

        while (true) {
            wait(mutex);
            read_count++;
            if (read_count == 1)
                wait(rw_mutex);
            signal(mutex);
            . . .
            /* reading is performed */
            . . .
            wait(mutex);
            read_count--;
            if (read_count == 0)
                signal(rw_mutex);
            signal(mutex);
        }
    
```

Figure 7.4 The structure of a reader process.

process wishes only to read shared data, it requests the reader–writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

7.1.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.5). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need

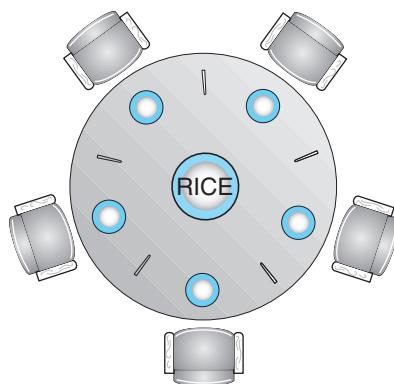


Figure 7.5 The situation of the dining philosophers.

```

        while (true) {
            wait(chopstick[i]);
            wait(chopstick[(i+1) % 5]);
            . . .
            /* eat for a while */
            . . .
            signal(chopstick[i]);
            signal(chopstick[(i+1) % 5]);
            . . .
            /* think for awhile */
            . . .
        }
    
```

Figure 7.6 The structure of philosopher *i*.

to allocate several resources among several processes in a deadlock-free and starvation-free manner.

7.1.3.1 Semaphore Solution

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. She releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 7.6.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are the following:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility

that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

7.1.3.2 Monitor Solution

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher i can set the variable `state[i] = EATING` only if her two neighbors are not eating: (`state[(i+4) % 5] != EATING`) and (`state[(i+1) % 5] != EATING`).

We also need to declare

```
condition self[5];
```

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor `DiningPhilosophers`, whose definition is shown in Figure 7.7. Each philosopher, before starting to eat, must invoke the operation `pickup()`. This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown()` operation. Thus, philosopher i must invoke the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. As we already noted, however, it is possible for a philosopher to starve to death. We do not present a solution to this problem but rather leave it as an exercise for you.

7.2 Synchronization within the Kernel

We next describe the synchronization mechanisms provided by the Windows and Linux operating systems. These two operating systems provide good examples of different approaches to synchronizing the kernel, and as you will

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Figure 7.7 A monitor solution to the dining-philosophers problem.

see, the synchronization mechanisms available in these systems differ in subtle yet significant ways.

7.2.1 Synchronization in Windows

The Windows operating system is a multithreaded kernel that provides support for real-time applications and multiple processors. When the Windows kernel accesses a global resource on a single-processor system, it temporarily masks interrupts for all interrupt handlers that may also access the global resource. On a multiprocessor system, Windows protects access to global resources using spinlocks, although the kernel uses spinlocks only to protect short code segments. Furthermore, for reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows provides **dispatcher objects**. Using a dispatcher object, threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. Semaphores behave as described in Section 6.6. **Events** are similar to condition variables; that is, they may notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired.

Dispatcher objects may be in either a signaled state or a nonsignaled state. An object in a **signaled state** is available, and a thread will not block when acquiring the object. An object in a **nonsignaled state** is not available, and a thread will block when attempting to acquire the object. We illustrate the state transitions of a mutex lock dispatcher object in Figure 7.8.

A relationship exists between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object. If so, the kernel moves one thread—or possibly more—from the waiting state to the ready state, where they can resume executing. The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object for which each thread is waiting. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be “owned” by only a single thread. For an event object, the kernel will select all threads that are waiting for the event.

We can use a mutex lock as an illustration of dispatcher objects and thread states. If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object. When the mutex moves to the signaled state (because another thread has released the lock on the mutex), the thread waiting at the front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

A **critical-section object** is a user-mode mutex that can often be acquired and released without kernel intervention. On a multiprocessor system, a critical-section object first uses a spinlock while waiting for the other thread to release the object. If it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU. Critical-section objects are particularly efficient because the kernel mutex is allocated only when there is contention for the object. In practice, there is very little contention, so the savings are significant.

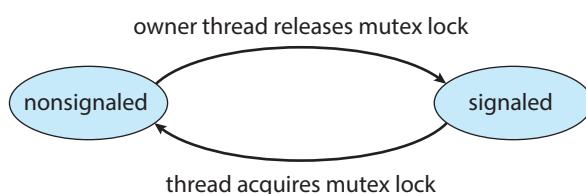


Figure 7.8 Mutex dispatcher object.

We provide a programming project at the end of this chapter that uses mutex locks and semaphores in the Windows API.

7.2.2 Synchronization in Linux

Prior to Version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel.

Linux provides several different mechanisms for synchronization in the kernel. As most computer architectures provide instructions for atomic versions of simple math operations, the simplest synchronization technique within the Linux kernel is an atomic integer, which is represented using the opaque data type `atomic_t`. As the name implies, all math operations using atomic integers are performed without interruption. To illustrate, consider a program that consists of an atomic integer counter and an integer value.

```
atomic_t counter;
int value;
```

The following code illustrates the effect of performing various atomic operations:

<i>Atomic Operation</i>	<i>Effect</i>
<code>atomic_set(&counter,5);</code>	<code>counter = 5</code>
<code>atomic_add(10,&counter);</code>	<code>counter = counter + 10</code>
<code>atomic_sub(4,&counter);</code>	<code>counter = counter - 4</code>
<code>atomic_inc(&counter);</code>	<code>counter = counter + 1</code>
<code>value = atomic_read(&counter);</code>	<code>value = 12</code>

Atomic integers are particularly efficient in situations where an integer variable—such as a counter—needs to be updated, since atomic operations do not require the overhead of locking mechanisms. However, their use is limited to these sorts of scenarios. In situations where there are several variables contributing to a possible race condition, more sophisticated locking tools must be used.

Mutex locks are available in Linux for protecting critical sections within the kernel. Here, a task must invoke the `mutex_lock()` function prior to entering a critical section and the `mutex_unlock()` function after exiting the critical section. If the mutex lock is unavailable, a task calling `mutex_lock()` is put into a sleep state and is awakened when the lock's owner invokes `mutex_unlock()`.

Linux also provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, such as embedded systems with only a single processing core, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel pre-emption. That is, on systems with a single processing core, rather than holding a spinlock, the kernel disables kernel pre-emption; and rather than releasing the spinlock, it enables kernel pre-emption. This is summarized below:

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

In the Linux kernel, both spinlocks and mutex locks are *nonrecursive*, which means that if a thread has acquired one of these locks, it cannot acquire the same lock a second time without first releasing the lock. Otherwise, the second attempt at acquiring the lock will block.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple system calls—`preempt_disable()` and `preempt_enable()`—for disabling and enabling kernel preemption. The kernel is not preemptible, however, if a task running in the kernel is holding a lock. To enforce this rule, each task in the system has a `thread_info` structure containing a counter, `preempt_count`, to indicate the number of locks being held by the task. When a lock is acquired, `preempt_count` is incremented. It is decremented when a lock is released. If the value of `preempt_count` for the task currently running in the kernel is greater than 0, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is 0, the kernel can safely be interrupted (assuming there are no outstanding calls to `preempt_disable()`).

Spinlocks—along with enabling and disabling kernel preemption—are used in the kernel only when a lock (or disabling kernel preemption) is held for a short duration. When a lock must be held for a longer period, semaphores or mutex locks are appropriate for use.

7.3 POSIX Synchronization

The synchronization methods discussed in the preceding section pertain to synchronization within the kernel and are therefore available only to kernel developers. In contrast, the POSIX API is available for programmers at the user level and is not part of any particular operating-system kernel. (Of course, it must ultimately be implemented using tools provided by the host operating system.)

In this section, we cover mutex locks, semaphores, and condition variables that are available in the Pthreads and POSIX APIs. These APIs are widely used for thread creation and synchronization by developers on UNIX, Linux, and macOS systems.

7.3.1 POSIX Mutex Locks

Mutex locks represent the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Pthreads uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init()` function. The first parameter is a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes. This is illustrated below:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

The mutex is acquired and released with the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`. The following code illustrates protecting a critical section with mutex locks:

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

7.3.2 POSIX Semaphores

Many systems that implement Pthreads also provide semaphores, although semaphores are not part of the POSIX standard and instead belong to the POSIX SEM extension. POSIX specifies two types of semaphores—[named](#) and [unnamed](#). Fundamentally, the two are quite similar, but they differ in terms of how they are created and shared between processes. Because both techniques are common, we discuss both here. Beginning with Version 2.6 of the kernel, Linux systems provide support for both named and unnamed semaphores.

7.3.2.1 POSIX Named Semaphores

The function `sem_open()` is used to create and open a POSIX named semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

In this instance, we are naming the semaphore SEM. The `O_CREAT` flag indicates that the semaphore will be created if it does not already exist. Additionally, the semaphore has read and write access for other processes (via the parameter `0666`) and is initialized to 1.

The advantage of named semaphores is that multiple unrelated processes can easily use a common semaphore as a synchronization mechanism by

simply referring to the semaphore's name. In the example above, once the semaphore SEM has been created, subsequent calls to `sem_open()` (with the same parameters) by other processes return a descriptor to the existing semaphore.

In Section 6.6, we described the classic `wait()` and `signal()` semaphore operations. POSIX declares these operations `sem_wait()` and `sem_post()`, respectively. The following code sample illustrates protecting a critical section using the named semaphore created above:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Both Linux and macOS systems provide POSIX named semaphores.

7.3.2.2 POSIX Unnamed Semaphores

An unnamed semaphore is created and initialized using the `sem_init()` function, which is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

and is illustrated in the following programming example:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

In this example, by passing the flag 0, we are indicating that this semaphore can be shared only by threads belonging to the process that created the semaphore. (If we supplied a nonzero value, we could allow the semaphore to be shared between separate processes by placing it in a region of shared memory.) In addition, we initialize the semaphore to the value 1.

POSIX unnamed semaphores use the same `sem_wait()` and `sem_post()` operations as named semaphores. The following code sample illustrates protecting a critical section using the unnamed semaphore created above:

```

/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);

```

Just like mutex locks, all semaphore functions return 0 when successful and nonzero when an error condition occurs.

7.3.3 POSIX Condition Variables

Condition variables in Pthreads behave similarly to those described in Section 6.7. However, in that section, condition variables are used within the context of a monitor, which provides a locking mechanism to ensure data integrity. Since Pthreads is typically used in C programs—and since C does not have a monitor—we accomplish locking by associating a condition variable with a mutex lock.

Condition variables in Pthreads use the `pthread_cond_t` data type and are initialized using the `pthread_cond_init()` function. The following code creates and initializes a condition variable as well as its associated mutex lock:

```

pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);

```

The `pthread_cond_wait()` function is used for waiting on a condition variable. The following code illustrates how a thread can wait for the condition `a == b` to become true using a Pthread condition variable:

```

pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);

```

The mutex lock associated with the condition variable must be locked before the `pthread_cond_wait()` function is called, since it is used to protect the data in the conditional clause from a possible race condition. Once this lock is acquired, the thread can check the condition. If the condition is not true, the thread then invokes `pthread_cond_wait()`, passing the mutex lock and the condition variable as parameters. Calling `pthread_cond_wait()` releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true. (To protect against program errors, it is important to place the conditional clause within a loop so that the condition is rechecked after being signaled.)

A thread that modifies the shared data can invoke the `pthread_cond_signal()` function, thereby signaling one thread waiting on the condition variable. This is illustrated below:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

It is important to note that the call to `pthread_cond_signal()` does not release the mutex lock. It is the subsequent call to `pthread_mutex_unlock()` that releases the mutex. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to `pthread_cond_wait()`.

We provide several programming problems and projects at the end of this chapter that use Pthreads mutex locks and condition variables, as well as POSIX semaphores.

7.4 Synchronization in Java

The Java language and its API have provided rich support for thread synchronization since the origins of the language. In this section, we first cover Java monitors, Java's original synchronization mechanism. We then cover three additional mechanisms that were introduced in Release 1.5: reentrant locks, semaphores, and condition variables. We include these because they represent the most common locking and synchronization mechanisms. However, the Java API provides many features that we do not cover in this text—for example, support for atomic variables and the CAS instruction—and we encourage interested readers to consult the bibliography for more information.

7.4.1 Java Monitors

Java provides a monitor-like concurrency mechanism for thread synchronization. We illustrate this mechanism with the `BoundedBuffer` class (Figure 7.9), which implements a solution to the bounded-buffer problem wherein the producer and consumer invoke the `insert()` and `remove()` methods, respectively.

Every object in Java has associated with it a single lock. When a method is declared to be synchronized, calling the method requires owning the lock for the object. We declare a synchronized method by placing the `synchronized` keyword in the method definition, such as with the `insert()` and `remove()` methods in the `BoundedBuffer` class.

Invoking a synchronized method requires owning the lock on an object instance of `BoundedBuffer`. If the lock is already owned by another thread, the thread calling the synchronized method blocks and is placed in the `entry set` for the object's lock. The entry set represents the set of threads waiting for the lock to become available. If the lock is available when a synchronized method is called, the calling thread becomes the owner of the object's lock and can enter the method. The lock is released when the thread exits the method. If the entry set for the lock is not empty when the lock is released, the JVM

```

public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

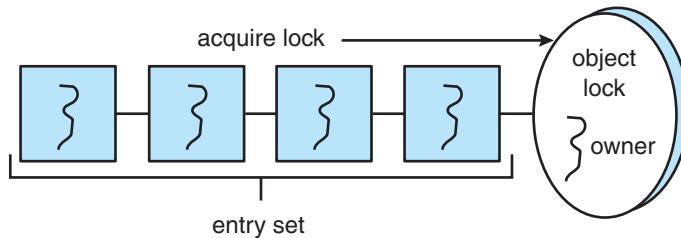
    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}

```

Figure 7.9 Bounded buffer using Java synchronization.

arbitrarily selects a thread from this set to be the owner of the lock. (When we say “arbitrarily,” we mean that the specification does not require that threads in this set be organized in any particular order. However, in practice, most virtual machines order threads in the entry set according to a FIFO policy.) Figure 7.10 illustrates how the entry set operates.

In addition to having a lock, every object also has associated with it a **wait set** consisting of a set of threads. This wait set is initially empty. When a thread enters a **synchronized** method, it owns the lock for the object. However, this thread may determine that it is unable to continue because a certain condition

**Figure 7.10** Entry set for a lock.

BLOCK SYNCHRONIZATION

The amount of time between when a lock is acquired and when it is released is defined as the **scope** of the lock. A **synchronized** method that has only a small percentage of its code manipulating shared data may yield a scope that is too large. In such an instance, it may be better to synchronize only the block of code that manipulates shared data than to synchronize the entire method. Such a design results in a smaller lock scope. Thus, in addition to declaring **synchronized** methods, Java also allows block synchronization, as illustrated below. Only the access to the critical-section code requires ownership of the object lock for the **this** object.

```
public void someMethod() {
    /* non-critical section */

    synchronized(this) {
        /* critical section */
    }

    /* remainder section */
}
```

has not been met. That will happen, for example, if the producer calls the **insert()** method and the buffer is full. The thread then will release the lock and wait until the condition that will allow it to continue is met.

When a thread calls the **wait()** method, the following happens:

1. The thread releases the lock for the object.
2. The state of the thread is set to blocked.
3. The thread is placed in the wait set for the object.

Consider the example in Figure 7.11. If the producer calls the **insert()** method and sees that the buffer is full, it calls the **wait()** method. This call releases the lock, blocks the producer, and puts the producer in the wait set for the object. Because the producer has released the lock, the consumer ultimately enters the **remove()** method, where it frees space in the buffer for the producer. Figure 7.12 illustrates the entry and wait sets for a lock. (Note that although **wait()** can throw an **InterruptedException**, we choose to ignore it for code clarity and simplicity.)

How does the consumer thread signal that the producer may now proceed? Ordinarily, when a thread exits a **synchronized** method, the departing thread releases only the lock associated with the object, possibly removing a thread from the entry set and giving it ownership of the lock. However, at the end of the **insert()** and **remove()** methods, we have a call to the method **notify()**. The call to **notify()**:

1. Picks an arbitrary thread T from the list of threads in the wait set

```

/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

notify();
}

/* Consumers call this method */
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
}

notify();

return item;
}

```

Figure 7.11 `insert()` and `remove()` methods using `wait()` and `notify()`.

2. Moves T from the wait set to the entry set
3. Sets the state of T from blocked to runnable

T is now eligible to compete for the lock with the other threads. Once T has regained control of the lock, it returns from calling `wait()`, where it may check the value of `count` again. (Again, the selection of an *arbitrary* thread is according to the Java specification; in practice, most Java virtual machines order threads in the wait set according to a FIFO policy.)

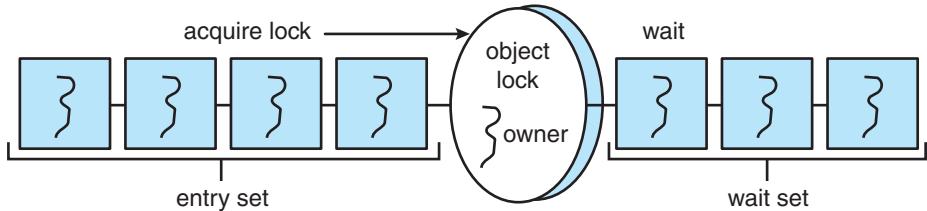


Figure 7.12 Entry and wait sets.

Next, we describe the `wait()` and `notify()` methods in terms of the methods shown in Figure 7.11. We assume that the buffer is full and the lock for the object is available.

- The producer calls the `insert()` method, sees that the lock is available, and enters the method. Once in the method, the producer determines that the buffer is full and calls `wait()`. The call to `wait()` releases the lock for the object, sets the state of the producer to blocked, and puts the producer in the wait set for the object.
- The consumer ultimately calls and enters the `remove()` method, as the lock for the object is now available. The consumer removes an item from the buffer and calls `notify()`. Note that the consumer still owns the lock for the object.
- The call to `notify()` removes the producer from the wait set for the object, moves the producer to the entry set, and sets the producer's state to runnable.
- The consumer exits the `remove()` method. Exiting this method releases the lock for the object.
- The producer tries to reacquire the lock and is successful. It resumes execution from the call to `wait()`. The producer tests the while loop, determines that room is available in the buffer, and proceeds with the remainder of the `insert()` method. If no thread is in the wait set for the object, the call to `notify()` is ignored. When the producer exits the method, it releases the lock for the object.

The `synchronized`, `wait()`, and `notify()` mechanisms have been part of Java since its origins. However, later revisions of the Java API introduced much more flexible and robust locking mechanisms, some of which we examine in the following sections.

7.4.2 Reentrant Locks

Perhaps the simplest locking mechanism available in the API is the `ReentrantLock`. In many ways, a `ReentrantLock` acts like the `synchronized` statement described in Section 7.4.1: a `ReentrantLock` is owned by a single thread and is used to provide mutually exclusive access to a shared resource. However, the `ReentrantLock` provides several additional features, such as setting a *fairness* parameter, which favors granting the lock to the longest-waiting thread. (Recall

that the specification for the JVM does not indicate that threads in the wait set for an object lock are to be ordered in any specific fashion.)

A thread acquires a `ReentrantLock` lock by invoking its `lock()` method. If the lock is available—or if the thread invoking `lock()` already owns it, which is why it is termed *reentrant*—`lock()` assigns the invoking thread lock ownership and returns control. If the lock is unavailable, the invoking thread blocks until it is ultimately assigned the lock when its owner invokes `unlock()`. `ReentrantLock` implements the `Lock` interface; it is used as follows:

```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```

The programming idiom of using `try` and `finally` requires a bit of explanation. If the lock is acquired via the `lock()` method, it is important that the lock be similarly released. By enclosing `unlock()` in a `finally` clause, we ensure that the lock is released once the critical section completes or if an exception occurs within the `try` block. Notice that we do not place the call to `lock()` within the `try` clause, as `lock()` does not throw any checked exceptions. Consider what happens if we place `lock()` within the `try` clause and an unchecked exception occurs when `lock()` is invoked (such as `OutOfMemoryError`): The `finally` clause triggers the call to `unlock()`, which then throws the unchecked `IllegalMonitorStateException`, as the lock was never acquired. This `IllegalMonitorStateException` replaces the unchecked exception that occurred when `lock()` was invoked, thereby obscuring the reason why the program initially failed.

Whereas a `ReentrantLock` provides mutual exclusion, it may be too conservative a strategy if multiple threads only read, but do not write, shared data. (We described this scenario in Section 7.1.2.) To address this need, the Java API also provides a `ReentrantReadWriteLock`, which is a lock that allows multiple concurrent readers but only one writer.

7.4.3 Semaphores

The Java API also provides a counting semaphore, as described in Section 6.6. The constructor for the semaphore appears as

```
Semaphore(int value);
```

where `value` specifies the initial value of the semaphore (a negative value is allowed). The `acquire()` method throws an `InterruptedException` if the acquiring thread is interrupted. The following example illustrates using a semaphore for mutual exclusion:

```

Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}

```

Notice that we place the call to `release()` in the `finally` clause to ensure that the semaphore is released.

7.4.4 Condition Variables

The last utility we cover in the Java API is the condition variable. Just as the `ReentrantLock` is similar to Java's `synchronized` statement, condition variables provide functionality similar to the `wait()` and `notify()` methods. Therefore, to provide mutual exclusion, a condition variable must be associated with a reentrant lock.

We create a condition variable by first creating a `ReentrantLock` and invoking its `newCondition()` method, which returns a `Condition` object representing the condition variable for the associated `ReentrantLock`. This is illustrated in the following statements:

```

Lock key = new ReentrantLock();
Condition condVar = key.newCondition();

```

Once the condition variable has been obtained, we can invoke its `await()` and `signal()` methods, which function in the same way as the `wait()` and `signal()` commands described in Section 6.7.

Recall that with monitors as described in Section 6.7, the `wait()` and `signal()` operations can be applied to *named* condition variables, allowing a thread to wait for a specific condition or to be notified when a specific condition has been met. At the language level, Java does not provide support for named condition variables. Each Java monitor is associated with just one unnamed condition variable, and the `wait()` and `notify()` operations described in Section 7.4.1 apply only to this single condition variable. When a Java thread is awakened via `notify()`, it receives no information as to why it was awakened; it is up to the reactivated thread to check for itself whether the condition for which it was waiting has been met. Condition variables remedy this by allowing a specific thread to be notified.

We illustrate with the following example: Suppose we have five threads, numbered 0 through 4, and a shared variable `turn` indicating which thread's turn it is. When a thread wishes to do work, it calls the `doWork()` method in Figure 7.13, passing its thread number. Only the thread whose value of `threadNumber` matches the value of `turn` can proceed; other threads must wait their turn.

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
         */

        /**
         * Now signal to the next thread.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}
```

Figure 7.13 Example using Java condition variables.

We also must create a `ReentrantLock` and five condition variables (representing the conditions the threads are waiting for) to signal the thread whose turn is next. This is shown below:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition();
```

When a thread enters `doWork()`, it invokes the `await()` method on its associated condition variable if its `threadNumber` is not equal to `turn`, only to resume when it is signaled by another thread. After a thread has completed its work, it signals the condition variable associated with the thread whose turn follows.

It is important to note that `doWork()` does not need to be declared synchronized, as the `ReentrantLock` provides mutual exclusion. When a thread

invokes `await()` on the condition variable, it releases the associated `ReentrantLock`, allowing another thread to acquire the mutual exclusion lock. Similarly, when `signal()` is invoked, only the condition variable is signaled; the lock is released by invoking `unlock()`.

7.5 Alternative Approaches

With the emergence of multicore systems has come increased pressure to develop concurrent applications that take advantage of multiple processing cores. However, concurrent applications present an increased risk of race conditions and liveness hazards such as deadlock. Traditionally, techniques such as mutex locks, semaphores, and monitors have been used to address these issues, but as the number of processing cores increases, it becomes increasingly difficult to design multithreaded applications that are free from race conditions and deadlock. In this section, we explore various features provided in both programming languages and hardware that support the design of thread-safe concurrent applications.

7.5.1 Transactional Memory

Quite often in computer science, ideas from one area of study can be used to solve problems in other areas. The concept of **transactional memory** originated in database theory, for example, yet it provides a strategy for process synchronization. A **memory transaction** is a sequence of memory read–write operations that are atomic. If all operations in a transaction are completed, the memory transaction is committed. Otherwise, the operations must be aborted and rolled back. The benefits of transactional memory can be obtained through features added to a programming language.

Consider an example. Suppose we have a function `update()` that modifies shared data. Traditionally, this function would be written using mutex locks (or semaphores) such as the following:

```
void update ()
{
    acquire();

    /* modify shared data */

    release();
}
```

However, using synchronization mechanisms such as mutex locks and semaphores involves many potential problems, including deadlock. Additionally, as the number of threads increases, traditional locking doesn't scale as well, because the level of contention among threads for lock ownership becomes very high.

As an alternative to traditional locking methods, new features that take advantage of transactional memory can be added to a programming language. In our example, suppose we add the construct `atomic{S}`, which ensures that

the operations in `S` execute as a transaction. This allows us to rewrite the `update()` function as follows:

```
void update ()
{
    atomic {
        /* modify shared data */
    }
}
```

The advantage of using such a mechanism rather than locks is that the transactional memory system—not the developer—is responsible for guaranteeing atomicity. Additionally, because no locks are involved, deadlock is not possible. Furthermore, a transactional memory system can identify which statements in atomic blocks can be executed concurrently, such as concurrent read access to a shared variable. It is, of course, possible for a programmer to identify these situations and use reader–writer locks, but the task becomes increasingly difficult as the number of threads within an application grows.

Transactional memory can be implemented in either software or hardware. **Software transactional memory (STM)**, as the name suggests, implements transactional memory exclusively in software—no special hardware is needed. STM works by inserting instrumentation code inside transaction blocks. The code is inserted by a compiler and manages each transaction by examining where statements may run concurrently and where specific low-level locking is required. **Hardware transactional memory (HTM)** uses hardware cache hierarchies and cache coherency protocols to manage and resolve conflicts involving shared data residing in separate processors' caches. HTM requires no special code instrumentation and thus has less overhead than STM. However, HTM does require that existing cache hierarchies and cache coherency protocols be modified to support transactional memory.

Transactional memory has existed for several years without widespread implementation. However, the growth of multicore systems and the associated emphasis on concurrent and parallel programming have prompted a significant amount of research in this area on the part of both academics and commercial software and hardware vendors.

7.5.2 OpenMP

In Section 4.5.2, we provided an overview of OpenMP and its support of parallel programming in a shared-memory environment. Recall that OpenMP includes a set of compiler directives and an API. Any code following the compiler directive `#pragma omp parallel` is identified as a parallel region and is performed by a number of threads equal to the number of processing cores in the system. The advantage of OpenMP (and similar tools) is that thread creation and management are handled by the OpenMP library and are not the responsibility of application developers.

Along with its `#pragma omp parallel` compiler directive, OpenMP provides the compiler directive `#pragma omp critical`, which specifies the code region following the directive as a critical section in which only one thread may be active at a time. In this way, OpenMP provides support for ensuring that threads do not generate race conditions.

As an example of the use of the critical-section compiler directive, first assume that the shared variable counter can be modified in the update() function as follows:

```
void update(int value)
{
    counter += value;
}
```

If the update() function can be part of—or invoked from—a parallel region, a race condition is possible on the variable counter.

The critical-section compiler directive can be used to remedy this race condition and is coded as follows:

```
void update(int value)
{
    #pragma omp critical
    {
        counter += value;
    }
}
```

The critical-section compiler directive behaves much like a binary semaphore or mutex lock, ensuring that only one thread at a time is active in the critical section. If a thread attempts to enter a critical section when another thread is currently active in that section (that is, *owns* the section), the calling thread is blocked until the owner thread exits. If multiple critical sections must be used, each critical section can be assigned a separate name, and a rule can specify that no more than one thread may be active in a critical section of the same name simultaneously.

An advantage of using the critical-section compiler directive in OpenMP is that it is generally considered easier to use than standard mutex locks. However, a disadvantage is that application developers must still identify possible race conditions and adequately protect shared data using the compiler directive. Additionally, because the critical-section compiler directive behaves much like a mutex lock, deadlock is still possible when two or more critical sections are identified.

7.5.3 Functional Programming Languages

Most well-known programming languages—such as C, C++, Java, and C#—are known as **imperative** (or **procedural**) languages. Imperative languages are used for implementing algorithms that are state-based. In these languages, the flow of the algorithm is crucial to its correct operation, and state is represented with variables and other data structures. Of course, program state is mutable, as variables may be assigned different values over time.

With the current emphasis on concurrent and parallel programming for multicore systems, there has been greater focus on **functional** programming languages, which follow a programming paradigm much different from that offered by imperative languages. The fundamental difference between imperative and functional languages is that functional languages do not maintain state. That is, once a variable has been defined and assigned a value, its value

is immutable—it cannot change. Because functional languages disallow mutable state, they need not be concerned with issues such as race conditions and deadlocks. Essentially, most of the problems addressed in this chapter are nonexistent in functional languages.

Several functional languages are presently in use, and we briefly mention two of them here: Erlang and Scala. The Erlang language has gained significant attention because of its support for concurrency and the ease with which it can be used to develop applications that run on parallel systems. Scala is a functional language that is also object-oriented. In fact, much of the syntax of Scala is similar to the popular object-oriented languages Java and C#. Readers interested in Erlang and Scala, and in further details about functional languages in general, are encouraged to consult the bibliography at the end of this chapter for additional references.

7.6 Summary

- Classic problems of process synchronization include the bounded-buffer, readers–writers, and dining-philosophers problems. Solutions to these problems can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, monitors, and condition variables.
- Windows uses dispatcher objects as well as events to implement process synchronization tools.
- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.
- The POSIX API provides mutex locks, semaphores, and condition variables. POSIX provides two forms of semaphores: named and unnamed. Several unrelated processes can easily access the same named semaphore by simply referring to its name. Unnamed semaphores cannot be shared as easily, and require placing the semaphore in a region of shared memory.
- Java has a rich library and API for synchronization. Available tools include monitors (which are provided at the language level) as well as reentrant locks, semaphores, and condition variables (which are supported by the API).
- Alternative approaches to solving the critical-section problem include transactional memory, OpenMP, and functional languages. Functional languages are particularly intriguing, as they offer a different programming paradigm from procedural languages. Unlike procedural languages, functional languages do not maintain state and therefore are generally immune from race conditions and critical sections.

Practice Exercises

- 7.1 Explain why Windows and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, and condition variables. In each case, explain why the mechanism is needed.

- 7.2 Windows provides a lightweight synchronization tool called **slim reader–writer** locks. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.
- 7.3 Describe what changes would be necessary to the producer and consumer processes in Figure 7.1 and Figure 7.2 so that a mutex lock could be used instead of a binary semaphore.
- 7.4 Describe how deadlock is possible with the dining-philosophers problem.
- 7.5 Explain the difference between signaled and non-signaled states with Windows dispatcher objects.
- 7.6 Assume `val` is an atomic integer in a Linux system. What is the value of `val` after the following operations have been completed?

```
atomic_set(&val,10);
atomic_sub(8,&val);
atomic_inc(&val);
atomic_inc(&val);
atomic_add(6,&val);
atomic_sub(3,&val);
```

Further Reading

Details of Windows synchronization can be found in [Solomon and Russinovich (2000)]. [Love (2010)] describes synchronization in the Linux kernel. [Hart (2005)] describes thread synchronization using Windows. [Breshears (2009)] and [Pacheco (2011)] provide detailed coverage of synchronization issues in relation to parallel programming. Details on using OpenMP can be found at <http://openmp.org>. Both [Oaks (2014)] and [Goetz et al. (2006)] contrast traditional synchronization and CAS-based strategies in Java.

Bibliography

- [Breshears (2009)] C. Breshears, *The Art of Concurrency*, O'Reilly & Associates (2009).
- [Goetz et al. (2006)] B. Goetz, T. Peirls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*, Addison-Wesley (2006).
- [Hart (2005)] J. M. Hart, *Windows System Programming*, Third Edition, Addison-Wesley (2005).
- [Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[**Oaks (2014)**] S. Oaks, *Java Performance—The Definitive Guide*, O'Reilly & Associates (2014).

[**Pacheco (2011)**] P. S. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann (2011).

[**Solomon and Russinovich (2000)**] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000*, Third Edition, Microsoft Press (2000).

Chapter 7 Exercises

- 7.7 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.
- 7.8 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.
- 7.9 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.
- 7.10 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 7.14 mainly suitable for small portions.
 - a. Explain why this is true.
 - b. Design a new scheme that is suitable for larger portions.
- 7.11 Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.
- 7.12 Explain why the call to the `lock()` method in a Java `ReentrantLock` is not placed in the `try` clause for exception handling, yet the call to the `unlock()` method is placed in a `finally` clause.
- 7.13 Explain the difference between software and hardware transactional memory.

Programming Problems

- 7.14** Exercise 3.20 required you to design a PID manager that allocated a unique process identifier to each process. Exercise 4.28 required you to modify your solution to Exercise 3.20 by writing a program that created a number of threads that requested and released process identifiers. Using mutex locks, modify your solution to Exercise 4.28 by ensuring that the data structure used to represent the availability of process identifiers is safe from race conditions.
- 7.15** In Exercise 4.27, you wrote a program to generate the Fibonacci sequence. The program required the parent thread to wait for the child thread to finish its execution before printing out the computed values. If we let the parent thread access the Fibonacci numbers as soon as they were computed by the child thread—rather than waiting for the child thread to terminate—what changes would be necessary to the solution for this exercise? Implement your modified solution.
- 7.16** The C program `stack-ptr.c` (available in the source-code download) contains an implementation of a stack using a linked list. An example of its use is as follows:

```
StackNode *top = NULL;
push(5, &top);
push(10, &top);
push(15, &top);

int value = pop(&top);
value = pop(&top);
value = pop(&top);
```

This program currently has a race condition and is not appropriate for a concurrent environment. Using Pthreads mutex locks (described in Section 7.3.1), fix the race condition.

- 7.17** Exercise 4.24 asked you to design a multithreaded program that estimated π using the Monte Carlo technique. In that exercise, you were asked to create a single thread that generated random points, storing the result in a global variable. Once that thread exited, the parent thread performed the calculation that estimated the value of π . Modify that program so that you create several threads, each of which generates random points and determines if the points fall within the circle. Each thread will have to update the global count of all points that fall within the circle. Protect against race conditions on updates to the shared global variable by using mutex locks.
- 7.18** Exercise 4.25 asked you to design a program using OpenMP that estimated π using the Monte Carlo technique. Examine your solution to that program looking for any possible race conditions. If you identify a race condition, protect against it using the strategy outlined in Section 7.5.2.
- 7.19** A **barrier** is a tool for synchronizing the activity of a number of threads. When a thread reaches a **barrier point**, it cannot proceed until all other

threads have reached this point as well. When the last thread reaches the barrier point, all threads are released and can resume concurrent execution.

Assume that the barrier is initialized to N —the number of threads that must wait at the barrier point:

```
init(N);
```

Each thread then performs some work until it reaches the barrier point:

```
/* do some work for awhile */

barrier_point();

/* do some work for awhile */
```

Using either the POSIX or Java synchronization tools described in this chapter, construct a barrier that implements the following API:

- `int init(int n)`—Initializes the barrier to the specified size.
- `int barrier_point(void)`—Identifies the barrier point. All threads are released from the barrier when the last thread reaches this point.

The return value of each function is used to identify error conditions. Each function will return 0 under normal operation and will return -1 if an error occurs. A testing harness is provided in the source-code download to test your implementation of the barrier.

Programming Projects

Project 1—Designing a Thread Pool

Thread pools were introduced in Section 4.5.1. When thread pools are used, a task is submitted to the pool and executed by a thread from the pool. Work is submitted to the pool using a queue, and an available thread removes work from the queue. If there are no available threads, the work remains queued until one becomes available. If there is no work, threads await notification until a task becomes available.

This project involves creating and managing a thread pool, and it may be completed using either Pthreads and POSIX synchronization or Java. Below we provide the details relevant to each specific technology.

I. POSIX

The POSIX version of this project will involve creating a number of threads using the Pthreads API as well as using POSIX mutex locks and semaphores for synchronization.

The Client

Users of the thread pool will utilize the following API:

- `void pool_init()`—Initializes the thread pool.
- `int pool_submit(void (*somefunction)(void *p), void *p)`—where `somefunction` is a pointer to the function that will be executed by a thread from the pool and `p` is a parameter passed to the function.
- `void pool_shutdown(void)`—Shuts down the thread pool once all tasks have completed.

We provide an example program `client.c` in the source code download that illustrates how to use the thread pool using these functions.

Implementation of the Thread Pool

In the source code download we provide the C source file `threadpool.c` as a partial implementation of the thread pool. You will need to implement the functions that are called by client users, as well as several additional functions that support the internals of the thread pool. Implementation will involve the following activities:

1. The `pool_init()` function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.
2. The `pool_submit()` function is partially implemented and currently places the function to be executed—as well as its data—into a task struct. The task struct represents work that will be completed by a thread in the pool. `pool_submit()` will add these tasks to the queue by invoking the `enqueue()` function, and worker threads will call `dequeue()` to retrieve work from the queue. The queue may be implemented statically (using arrays) or dynamically (using a linked list).

The `pool_init()` function has an `int` return value that is used to indicate if the task was successfully submitted to the pool (0 indicates success, 1 indicates failure). If the queue is implemented using arrays, `pool_init()` will return 1 if there is an attempt to submit work and the queue is full. If the queue is implemented as a linked list, `pool_init()` should always return 0 unless a memory allocation error occurs.

3. The `worker()` function is executed by each thread in the pool, where each thread will wait for available work. Once work becomes available, the thread will remove it from the queue and invoke `execute()` to run the specified function.

A semaphore can be used for notifying a waiting thread when work is submitted to the thread pool. Either named or unnamed semaphores may be used. Refer to Section 7.3.2 for further details on using POSIX semaphores.

4. A mutex lock is necessary to avoid race conditions when accessing or modifying the queue. (Section 7.3.1 provides details on Pthreads mutex locks.)
5. The `pool_shutdown()` function will cancel each worker thread and then wait for each thread to terminate by calling `pthread_join()`. Refer to Section 4.6.3 for details on POSIX thread cancellation. (The semaphore operation `sem_wait()` is a cancellation point that allows a thread waiting on a semaphore to be cancelled.)

Refer to the source-code download for additional details on this project. In particular, the `README` file describes the source and header files, as well as the `Makefile` for building the project.

II. Java

The Java version of this project may be completed using Java synchronization tools as described in Section 7.4. Synchronization may depend on either (a) monitors using `synchronized/wait()/notify()` (Section 7.4.1) or (b) semaphores and reentrant locks (Section 7.4.2 and Section 7.4.3). Java threads are described in Section 4.4.3.

Implementation of the Thread Pool

Your thread pool will implement the following API:

- `ThreadPool()`—Create a default-sized thread pool.
- `ThreadPool(int size)`—Create a thread pool of size `size`.
- `void add(Runnable task)`—Add a task to be performed by a thread in the pool.
- `void shutdown()`—Stop all threads in the pool.

We provide the Java source file `ThreadPool.java` as a partial implementation of the thread pool in the source code download. You will need to implement the methods that are called by client users, as well as several additional methods that support the internals of the thread pool. Implementation will involve the following activities:

1. The constructor will first create a number of idle threads that await work.
2. Work will be submitted to the pool via the `add()` method, which adds a task implementing the `Runnable` interface. The `add()` method will place the `Runnable` task into a queue (you may use an available structure from the Java API such as `java.util.List`).
3. Once a thread in the pool becomes available for work, it will check the queue for any `Runnable` tasks. If there is such a task, the idle thread will remove the task from the queue and invoke its `run()` method. If the queue is empty, the idle thread will wait to be notified when work

becomes available. (The `add()` method may implement notification using either `notify()` or semaphore operations when it places a `Runnable` task into the queue to possibly awaken an idle thread awaiting work.)

4. The `shutdown()` method will stop all threads in the pool by invoking their `interrupt()` method. This, of course, requires that `Runnable` tasks being executed by the thread pool check their interruption status (Section 4.6.3).

Refer to the source-code download for additional details on this project. In particular, the `README` file describes the Java source files, as well as further details on Java thread interruption.

Project 2—The Sleeping Teaching Assistant

A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

Using POSIX threads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

The Students and the TA

Using Pthreads (Section 4.4.1), begin by creating n students where each student will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.

Perhaps the best option for simulating students programming—as well as the TA providing help to a student—is to have the appropriate threads sleep for a random period of time.

Coverage of POSIX mutex locks and semaphores is provided in Section 7.3. Consult that section for details.

Project 3—The Dining-Philosophers Problem

In Section 7.1.3, we provide an outline of a solution to the dining-philosophers problem using monitors. This project involves implementing a solution to this problem using either POSIX mutex locks and condition variables or Java condition variables. Solutions will be based on the algorithm illustrated in Figure 7.7.

Both implementations will require creating five philosophers, each identified by a number 0 . . . 4. Each philosopher will run as a separate thread. Philosophers alternate between thinking and eating. To simulate both activities, have each thread sleep for a random period between one and three seconds.

I. POSIX

Thread creation using Pthreads is covered in Section 4.4.1. When a philosopher wishes to eat, she invokes the function

```
pickup_forks(int philosopher_number)
```

where `philosopher_number` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes

```
return_forks(int philosopher_number)
```

Your implementation will require the use of POSIX condition variables, which are covered in Section 7.3.

II. Java

When a philosopher wishes to eat, she invokes the method `takeForks(philosopherNumber)`, where `philosopherNumber` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes `returnForks(philosopherNumber)`.

Your solution will implement the following interface:

```
public interface DiningServer
{
    /* Called by a philosopher when it wishes to eat */
    public void takeForks(int philosopherNumber);

    /* Called by a philosopher when it is finished eating */
    public void returnForks(int philosopherNumber);
}
```

It will require the use of Java condition variables, which are covered in Section 7.4.4.

Project 4—The Producer–Consumer Problem

In Section 7.1.1, we presented a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 5.9 and 5.10. The solution presented in Section 7.1.1 uses three semaphores: `empty` and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual-exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores for `empty` and `full` and a mutex lock, rather than a binary semaphore, to represent `mutex`. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with the `empty`, `full`, and `mutex` structures. You can solve this problem using either Pthreads or the Windows API.

The Buffer

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a `typedef`). The array of `buffer_item` objects will be manipulated as a circular queue. The definition of `buffer_item`, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 7.14.

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figure 7.1 and Figure 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the `empty` and `full` semaphores.

The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main()` function will sleep for a period of time and, upon awaking, will terminate the application. The `main()` function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears in Figure 7.15.

```
#include "buffer.h"

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
    /* insert item into buffer
       return 0 if successful, otherwise
       return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
    /* remove an object from buffer
       placing it in item
       return 0 if successful, otherwise
       return -1 indicating an error condition */
}
```

Figure 7.14 Outline of buffer operations.

The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand()` function, which produces random integers between 0 and `RAND_MAX`. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears in Figure 7.16.

```
#include "buffer.h"

int main(int argc, char *argv[]) {
    /* 1. Get command line arguments argv[1], argv[2], argv[3] */
    /* 2. Initialize buffer */
    /* 3. Create producer thread(s) */
    /* 4. Create consumer thread(s) */
    /* 5. Sleep */
    /* 6. Exit */
}
```

Figure 7.15 Outline of skeleton program.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

void *producer(void *param) {
    buffer_item item;

    while (true) {
        /* sleep for a random period of time */
        sleep(...);
        /* generate a random number */
        item = rand();
        if (insert_item(item))
            fprintf("report error condition");
        else
            printf("producer produced %d\n",item);
    }
}

void *consumer(void *param) {
    buffer_item item;

    while (true) {
        /* sleep for a random period of time */
        sleep(...);
        if (remove_item(&item))
            fprintf("report error condition");
        else
            printf("consumer consumed %d\n",item);
    }
}
```

Figure 7.16 An outline of the producer and consumer threads.

As noted earlier, you can solve this problem using either Pthreads or the Windows API. In the following sections, we supply more information on each of these choices.

Pthreads Thread Creation and Synchronization

Creating threads using the Pthreads API is discussed in Section 4.4.1. Coverage of mutex locks and semaphores using Pthreads is provided in Section 7.3. Refer to those sections for specific instructions on Pthreads thread creation and synchronization.

Windows Threads

Section 4.4.2 discusses thread creation using the Windows API. Refer to that section for specific instructions on creating threads.

Windows Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 7.2.1. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

The first parameter refers to a security attribute for the mutex lock. By setting this attribute to `NULL`, we prevent any children of the process from creating this mutex lock to inherit the handle of the lock. The second parameter indicates whether the creator of the mutex lock is the lock's initial owner. Passing a value of `FALSE` indicates that the thread creating the mutex is not the initial owner. (We shall soon see how mutex locks are acquired.) The third parameter allows us to name the mutex. However, because we provide a value of `NULL`, we do not name the mutex. If successful, `CreateMutex()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

In Section 7.2.1, we identified dispatcher objects as being either *signaled* or *nonsignaled*. A signaled dispatcher object (such as a mutex lock) is available for ownership. Once it is acquired, it moves to the nonsignaled state. When it is released, it returns to signaled.

Mutex locks are acquired by invoking the `WaitForSingleObject()` function. The function is passed the `HANDLE` to the lock along with a flag indicating how long to wait. The following code demonstrates how the mutex lock created above can be acquired:

```
WaitForSingleObject(Mutex, INFINITE);
```

The parameter value `INFINITE` indicates that we will wait an infinite amount of time for the lock to become available. Other values could be used that would allow the calling thread to time out if the lock did not become available within a specified time. If the lock is in a signaled state, `WaitForSingleObject()` returns immediately, and the lock becomes nonsignaled. A lock is released (moves to the signaled state) by invoking `ReleaseMutex()`—for example, as follows:

```
ReleaseMutex(Mutex);
```

Windows Semaphores

Semaphores in the Windows API are dispatcher objects and thus use the same signaling mechanism as mutex locks. Semaphores are created as follows:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

The first and last parameters identify a security attribute and a name for the semaphore, similar to what we described for mutex locks. The second and third parameters indicate the initial value and maximum value of the semaphore. In this instance, the initial value of the semaphore is 1, and its maximum value is 5. If successful, `CreateSemaphore()` returns a HANDLE to the mutex lock; otherwise, it returns NULL.

Semaphores are acquired with the same `WaitForSingleObject()` function as mutex locks. We acquire the semaphore `Sem` created in this example by using the following statement:

```
WaitForSingleObject(Sem, INFINITE);
```

If the value of the semaphore is > 0 , the semaphore is in the signaled state and thus is acquired by the calling thread. Otherwise, the calling thread blocks indefinitely—as we are specifying INFINITE—until the semaphore returns to the signaled state.

The equivalent of the `signal()` operation for Windows semaphores is the `ReleaseSemaphore()` function. This function is passed three parameters:

1. The HANDLE of the semaphore
2. How much to increase the value of the semaphore
3. A pointer to the previous value of the semaphore

We can use the following statement to increase `Sem` by 1:

```
ReleaseSemaphore(Sem, 1, NULL);
```

Both `ReleaseSemaphore()` and `ReleaseMutex()` return a nonzero value if successful and 0 otherwise.

Deadlocks



In a multiprogramming environment, several threads may compete for a finite number of resources. A thread requests resources; if the resources are not available at that time, the thread enters a waiting state. Sometimes, a waiting thread can never again change state, because the resources it has requested are held by other waiting threads. This situation is called a **deadlock**. We discussed this issue briefly in Chapter 6 as a form of liveness failure. There, we defined deadlock as a situation in which *every process in a set of processes is waiting for an event that can be caused only by another process in the set*.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

In this chapter, we describe methods that application developers as well as operating-system programmers can use to prevent or deal with deadlocks. Although some applications can identify programs that may deadlock, operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs. Deadlock problems—as well as other liveness failures—are becoming more challenging as demand continues for increased concurrency and parallelism on multicore systems.

CHAPTER OBJECTIVES

- Illustrate how deadlock can occur when mutex locks are used.
- Define the four necessary conditions that characterize deadlock.
- Identify a deadlock situation in a resource allocation graph.
- Evaluate the four different approaches for preventing deadlocks.
- Apply the banker’s algorithm for deadlock avoidance.
- Apply the deadlock detection algorithm.
- Evaluate approaches for recovering from deadlock.

8.1 System Model

A system consists of a finite number of resources to be distributed among a number of competing threads. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as network interfaces and DVD drives) are examples of resource types. If a system has four CPUs, then the resource type *CPU* has four instances. Similarly, the resource type *network* may have two instances. If a thread requests an instance of a resource type, the allocation of *any* instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

The various synchronization tools discussed in Chapter 6, such as mutex locks and semaphores, are also system resources; and on contemporary computer systems, they are the most common sources of deadlock. However, definition is not a problem here. A lock is typically associated with a specific data structure—that is, one lock may be used to protect access to a queue, another to protect access to a linked list, and so forth. For that reason, each instance of a lock is typically assigned its own resource class.

Note that throughout this chapter we discuss kernel resources, but threads may use resources from other processes (for example, via interprocess communication), and those resource uses can also result in deadlock. Such deadlocks are not the concern of the kernel and thus not described here.

A thread must request a resource before using it and must release the resource after using it. A thread may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a thread cannot request two network interfaces if the system has only one.

Under the normal mode of operation, a thread may utilize a resource in only the following sequence:

1. **Request.** The thread requests the resource. If the request cannot be granted immediately (for example, if a mutex lock is currently held by another thread), then the requesting thread must wait until it can acquire the resource.
2. **Use.** The thread can operate on the resource (for example, if the resource is a mutex lock, the thread can access its critical section).
3. **Release.** The thread releases the resource.

The request and release of resources may be system calls, as explained in Chapter 2. Examples are the `request()` and `release()` of a device, `open()` and `close()` of a file, and `allocate()` and `free()` memory system calls. Similarly, as we saw in Chapter 6, request and release can be accomplished through the `wait()` and `signal()` operations on semaphores and through `acquire()` and `release()` of a mutex lock. For each use of a kernel-managed resource by a thread, the operating system checks to make sure that the thread has requested and has been allocated the resource. A system table records whether each resource is free or allocated. For each resource that is allocated,

the table also records the thread to which it is allocated. If a thread requests a resource that is currently allocated to another thread, it can be added to a queue of threads waiting for this resource.

A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources are typically logical (for example, mutex locks, semaphores, and files); however, other types of events may result in deadlocks, including reading from a network interface or the IPC (interprocess communication) facilities discussed in Chapter 3.

To illustrate a deadlocked state, we refer back to the dining-philosophers problem from Section 7.1.3. In this situation, resources are represented by chopsticks. If all the philosophers get hungry at the same time, and each philosopher grabs the chopstick on her left, there are no longer any available chopsticks. Each philosopher is then blocked waiting for her right chopstick to become available.

Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in Chapter 6 are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released. Otherwise, deadlock can occur, as described next.

8.2 Deadlock in Multithreaded Applications

Prior to examining how deadlock issues can be identified and managed, we first illustrate how deadlock can occur in a multithreaded Pthread program using POSIX mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created and initialized in the following code example:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two` run in the functions `do_work_one()` and `do_work_two()`, respectively, as shown in Figure 8.1.

In this example, `thread_one` attempts to acquire the mutex locks in the order (1) `first_mutex`, (2) `second_mutex`. At the same time, `thread_two` attempts to acquire the mutex locks in the order (1) `second_mutex`, (2) `first_mutex`. Deadlock is possible if `thread_one` acquires `first_mutex` while `thread_two` acquires `second_mutex`.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Figure 8.1 Deadlock example.

Note that, even though deadlock is possible, it will not occur if `thread_one` can acquire and release the mutex locks for `first_mutex` and `second_mutex` before `thread_two` attempts to acquire the locks. And, of course, the order in which the threads run depends on how they are scheduled by the CPU scheduler. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

8.2.1 Livelock

Livelock is another form of liveness failure. It is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons. Whereas deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, livelock occurs when a thread continuously attempts an action that fails. Livelock is similar to what sometimes happens when two people attempt to pass in a hallway: One moves to his right, the other to her left, still obstructing each other's progress. Then he moves to his left, and she moves

to her right, and so forth. They aren't blocked, but they aren't making any progress.

Livelock can be illustrated with the Pthreads `pthread_mutex_trylock()` function, which attempts to acquire a mutex lock without blocking. The code example in Figure 8.2 rewrites the example from Figure 8.1 so that it now uses `pthread_mutex_trylock()`. This situation can lead to livelock if `thread_one` acquires `first_mutex`, followed by `thread_two` acquiring `second_mutex`. Each thread then invokes `pthread_mutex_trylock()`, which fails, releases their respective locks, and repeats the same actions indefinitely.

Livelock typically occurs when threads retry failing operations at the same time. It thus can generally be avoided by having each thread retry the failing operation at random times. This is precisely the approach taken by Ethernet networks when a network collision occurs. Rather than trying to retransmit a packet immediately after a collision occurs, a host involved in a collision will *backoff* a random period of time before attempting to transmit again.

Livelock is less common than deadlock but nonetheless is a challenging issue in designing concurrent applications, and like deadlock, it may only occur under specific scheduling circumstances.

8.3 Deadlock Characterization

In the previous section we illustrated how deadlock could occur in multi-threaded programming using mutex locks. We now look more closely at conditions that characterize deadlock.

8.3.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- 1. Mutual exclusion.** At least one resource must be held in a nonsharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
- 2. Hold and wait.** A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
- 3. No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
- 4. Circular wait.** A set $\{T_0, T_1, \dots, T_n\}$ of waiting threads must exist such that T_0 is waiting for a resource held by T_1 , T_1 is waiting for a resource held by T_2, \dots, T_{n-1} is waiting for a resource held by T_n , and T_n is waiting for a resource held by T_0 .

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&first_mutex);
        if (pthread_mutex_trylock(&second_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&first_mutex);
    }

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&second_mutex);
    }

    pthread_exit(0);
}
```

Figure 8.2 Livelock example.

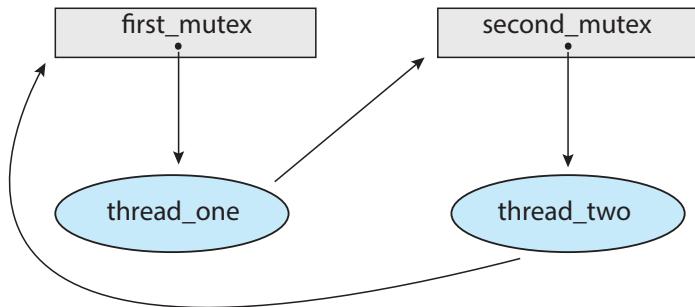


Figure 8.3 Resource-allocation graph for program in Figure 8.1.

conditions are not completely independent. We shall see in Section 8.5, however, that it is useful to consider each condition separately.

8.3.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E . The set of vertices V is partitioned into two different types of nodes: $T = \{T_1, T_2, \dots, T_n\}$, the set consisting of all the active threads in the system, and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.

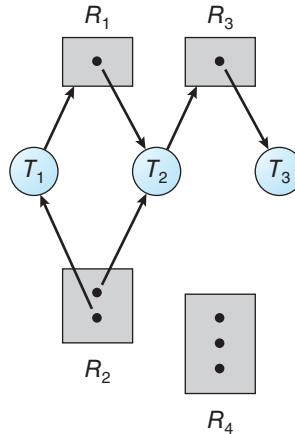
A directed edge from thread T_i to resource type R_j is denoted by $T_i \rightarrow R_j$; it signifies that thread T_i has requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to thread T_i is denoted by $R_j \rightarrow T_i$; it signifies that an instance of resource type R_j has been allocated to thread T_i . A directed edge $T_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow T_i$ is called an **assignment edge**.

Pictorially, we represent each thread T_i as a circle and each resource type R_j as a rectangle. As a simple example, the resource allocation graph shown in Figure 8.3 illustrates the deadlock situation from the program in Figure 8.1. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points only to the rectangle R_j , whereas an assignment edge must also designate one of the dots in the rectangle.

When thread T_i requests an instance of resource type R_j , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the thread no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure 8.4 depicts the following situation.

- The sets T , R , and E :
 - $T = \{T_1, T_2, T_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$

**Figure 8.4** Resource-allocation graph.

- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
- Resource instances:
 - One instance of resource type R_1
 - Two instances of resource type R_2
 - One instance of resource type R_3
 - Three instances of resource type R_4
- Thread states:
 - Thread T_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1 .
 - Thread T_2 is holding instances of R_1 and R_2 and is waiting for an instance of R_3 .
 - Thread T_3 is holding an instance of R_3 .

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each thread involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 8.4. Suppose that thread T_3 requests an instance of resource type R_2 . Since no resource instance is currently available, we add a request

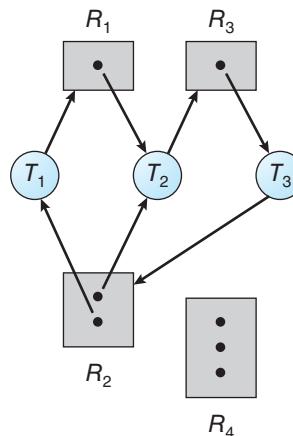


Figure 8.5 Resource-allocation graph with a deadlock.

edge $T_3 \rightarrow R_2$ to the graph (Figure 8.5). At this point, two minimal cycles exist in the system:

$$\begin{array}{ccccccc} T_1 & \rightarrow & R_1 & \rightarrow & T_2 & \rightarrow & R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1 \\ T_2 & \rightarrow & R_3 & \rightarrow & T_3 & \rightarrow & R_2 \rightarrow T_2 \end{array}$$

Threads T_1 , T_2 , and T_3 are deadlocked. Thread T_2 is waiting for the resource R_3 , which is held by thread T_3 . Thread T_3 is waiting for either thread T_1 or thread T_2 to release resource R_2 . In addition, thread T_1 is waiting for thread T_2 to release resource R_1 .

Now consider the resource-allocation graph in Figure 8.6. In this example, we also have a cycle:

$$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$$

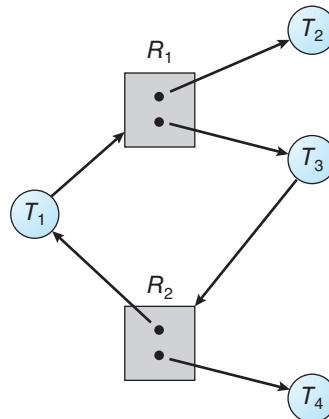


Figure 8.6 Resource-allocation graph with a cycle but no deadlock.

However, there is no deadlock. Observe that thread T_4 may release its instance of resource type R_2 . That resource can then be allocated to T_3 , breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system *may* or *may not* be in a deadlocked state. This observation is important when we deal with the deadlock problem.

8.4 Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.

The first solution is the one used by most operating systems, including Linux and Windows. It is then up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution. Some systems—such as databases—adopt the third solution, allowing deadlocks to occur and then managing the recovery.

Next, we elaborate briefly on the three methods for handling deadlocks. Then, in Section 8.5 through Section 8.8, we present detailed algorithms. Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions (Section 8.3.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 8.5.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the thread should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread. We discuss these schemes in Section 8.6.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from

the deadlock (if a deadlock has indeed occurred). We discuss these issues in Section 8.7 and Section 8.8.

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by threads that cannot run and because more and more threads, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned earlier. Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per month), the extra expense of the other methods may not seem worthwhile.

In addition, methods used to recover from other liveness conditions, such as livelock, may be used to recover from deadlock. In some circumstances, a system is suffering from a liveness failure but is not in a deadlocked state. We see this situation, for example, with a real-time thread running at the highest priority (or any thread running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

8.5 Deadlock Prevention

As we noted in Section 8.3.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

8.5.1 Mutual Exclusion

The mutual-exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several threads attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A thread never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable. For example, a mutex lock cannot be simultaneously shared by several threads.

8.5.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources. One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution. This is, of course,

impractical for most applications due to the dynamic nature of requesting resources.

An alternative protocol allows a thread to request resources only when it has none. A thread may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. For example, a thread may be allocated a mutex lock for its entire execution, yet only require it for a short duration. Second, starvation is possible. A thread that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other thread.

8.5.3 No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the thread is waiting. The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a thread requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other thread that is waiting for additional resources. If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread. If the resources are neither available nor held by a waiting thread, the requesting thread must wait. While it is waiting, some of its resources may be preempted, but only if another thread requests them. A thread can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions. It cannot generally be applied to such resources as mutex locks and semaphores, precisely the type of resources where deadlock occurs most commonly.

8.5.4 Circular Wait

The three options presented thus far for deadlock prevention are generally impractical in most situations. However, the fourth and final condition for deadlocks — the circular-wait condition — presents an opportunity for a practical solution by invalidating one of the necessary conditions. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.

To illustrate, we let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to

compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers. We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. For example, the lock ordering in the Pthread program shown in Figure 8.1 could be

$$\begin{aligned}F(\text{first_mutex}) &= 1 \\F(\text{second_mutex}) &= 5\end{aligned}$$

We can now consider the following protocol to prevent deadlocks: Each thread can request resources only in an increasing order of enumeration. That is, a thread can initially request an instance of a resource—say, R_i . After that, the thread can request an instance of resource R_j if and only if $F(R_j) > F(R_i)$. For example, using the function defined above, a thread that wants to use both `first_mutex` and `second_mutex` at the same time must first request `first_mutex` and then `second_mutex`. Alternatively, we can require that a thread requesting an instance of resource R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$. Note also that if several instances of the same resource type are needed, a *single* request for all of them must be issued.

If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of threads involved in the circular wait be $\{T_0, T_1, \dots, T_n\}$, where T_i is waiting for a resource R_i , which is held by thread T_{i+1} . (Modulo arithmetic is used on the indexes, so that T_n is waiting for a resource R_n held by T_0 .) Then, since thread T_{i+1} is holding resource R_i while requesting resource R_{i+1} , we must have $F(R_i) < F(R_{i+1})$ for all i . But this condition means that $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering. However, establishing a lock ordering can be difficult, especially on a system with hundreds—or thousands—of locks. To address this challenge, many Java developers have adopted the strategy of using the method `System.identityHashCode(Object)` (which returns the default hash code value of the `Object` parameter it has been passed) as the function for ordering lock acquisition.

It is also important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. For example, assume we have a function that transfers funds between two accounts. To prevent a race condition, each account has an associated mutex lock that is obtained from a `get_lock()` function such as that shown in Figure 8.7. Deadlock is possible if two threads simultaneously invoke the `transaction()` function, transposing different accounts. That is, one thread might invoke

```
transaction(checking_account, savings_account, 25.0)
```

and another might invoke

```
transaction(savings_account, checking_account, 50.0)
```

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);

    acquire(lock1);
    acquire(lock2);

    withdraw(from, amount);
    deposit(to, amount);

    release(lock2);
    release(lock1);
}
```

Figure 8.7 Deadlock example with lock ordering.

8.6 Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in Section 8.5, prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with resources R_1 and R_2 , the system might need to know that thread P will request first R_1 and then R_2 before releasing both resources, whereas thread Q will request R_2 and then R_1 . With this knowledge of the complete sequence of requests and releases for each thread, the system can decide for each request whether or not the thread should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each thread declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation *state* is defined by the number of available and allocated resources and the maximum demands of the threads. In the following sections, we explore two deadlock-avoidance algorithms.

LINUX LOCKDEP TOOL

Although ensuring that resources are acquired in the proper order is the responsibility of kernel and application developers, certain software can be used to verify that locks are acquired in the proper order. To detect possible deadlocks, Linux provides `lockdep`, a tool with rich functionality that can be used to verify locking order in the kernel. `lockdep` is designed to be enabled on a running kernel as it monitors usage patterns of lock acquisitions and releases against a set of rules for acquiring and releasing locks. Two examples follow, but note that `lockdep` provides significantly more functionality than what is described here:

- The order in which locks are acquired is dynamically maintained by the system. If `lockdep` detects locks being acquired out of order, it reports a possible deadlock condition.
- In Linux, spinlocks can be used in interrupt handlers. A possible source of deadlock occurs when the kernel acquires a spinlock that is also used in an interrupt handler. If the interrupt occurs while the lock is being held, the interrupt handler preempts the kernel code currently holding the lock and then spins while attempting to acquire the lock, resulting in deadlock. The general strategy for avoiding this situation is to disable interrupts on the current processor before acquiring a spinlock that is also used in an interrupt handler. If `lockdep` detects that interrupts are enabled while kernel code acquires a lock that is also used in an interrupt handler, it will report a possible deadlock scenario.

`lockdep` was developed to be used as a tool in developing or modifying code in the kernel and not to be used on production systems, as it can significantly slow down a system. Its purpose is to test whether software such as a new device driver or kernel module provides a possible source of deadlock. The designers of `lockdep` have reported that within a few years of its development in 2006, the number of deadlocks from system reports had been reduced by an order of magnitude.¹² Although `lockdep` was originally designed only for use in the kernel, recent revisions of this tool can now be used for detecting deadlocks in user applications using Pthreads mutex locks. Further details on the `lockdep` tool can be found at <https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt>.

8.6.1 Safe State

A state is *safe* if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of threads $\langle T_1, T_2, \dots, T_n \rangle$ is a safe sequence for the current allocation state if, for each T_i , the resource requests that T_i can still make can be satisfied by the currently available resources plus the resources held by all T_j , with $j < i$. In this situation, if the resources that T_i needs are not immediately available, then T_i can wait until all T_j have finished. When they have finished, T_i can obtain all of its

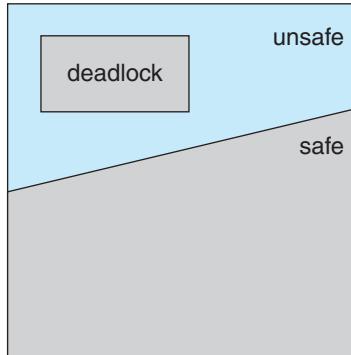


Figure 8.8 Safe, unsafe, and deadlocked state spaces.

needed resources, complete its designated task, return its allocated resources, and terminate. When T_i terminates, T_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 8.8). An unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent threads from requesting resources in such a way that a deadlock occurs. The behavior of the threads controls unsafe states.

To illustrate, consider a system with twelve resources and three threads: T_0 , T_1 , and T_2 . Thread T_0 requires ten resources, thread T_1 may need as many as four, and thread T_2 may need up to nine resources. Suppose that, at time t_0 , thread T_0 is holding five resources, thread T_1 is holding two resources, and thread T_2 is holding two resources. (Thus, there are three free resources.)

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	2

At time t_0 , the system is in a safe state. The sequence $\langle T_1, T_0, T_2 \rangle$ satisfies the safety condition. Thread T_1 can immediately be allocated all its resources and then return them (the system will then have five available resources); then thread T_0 can get all its resources and return them (the system will then have ten available resources); and finally thread T_2 can get all its resources and return them (the system will then have all twelve resources available).

A system can go from a safe state to an unsafe state. Suppose that, at time t_1 , thread T_2 requests and is allocated one more resource. The system is no longer in a safe state. At this point, only thread T_1 can be allocated all its resources. When it returns them, the system will have only four available resources. Since thread T_0 is allocated five resources but has a maximum of ten, it may request five more resources. If it does so, it will have to wait, because they are unavailable. Similarly, thread T_2 may request six additional resources and have to wait, resulting in a deadlock. Our mistake was in granting the request from thread T_2 for one more resource. If we had made T_2 wait until either of the other

threads had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a thread requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or the thread must wait. The request is granted only if the allocation leaves the system in a safe state.

In this scheme, if a thread requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

8.6.2 Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined in Section 8.3.2 for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**. A claim edge $T_i \rightarrow R_j$ indicates that thread T_i may request resource R_j at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When thread T_i requests resource R_j , the claim edge $T_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by T_i , the assignment edge $R_j \rightarrow T_i$ is reconverted to a claim edge $T_i \rightarrow R_j$.

Note that the resources must be claimed a priori in the system. That is, before thread T_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $T_i \rightarrow R_j$ to be added to the graph only if all the edges associated with thread T_i are claim edges.

Now suppose that thread T_i requests resource R_j . The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of threads in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, thread T_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 8.9. Suppose that T_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to T_2 , since this action will create a cycle in the graph (Figure 8.10). A cycle, as mentioned, indicates that the system is in an unsafe state. If T_1 requests R_2 , and T_2 requests R_1 , then a deadlock will occur.

8.6.3 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The

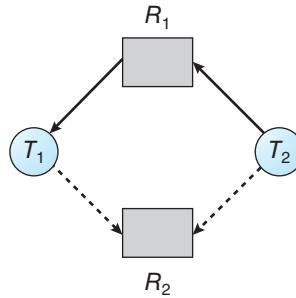


Figure 8.9 Resource-allocation graph for deadlock avoidance.

deadlock-avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm**. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the thread must wait until some other thread releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of threads in the system and m is the number of resource types:

- **Available.** A vector of length m indicates the number of available resources of each type. If $Available[j]$ equals k , then k instances of resource type R_j are available.

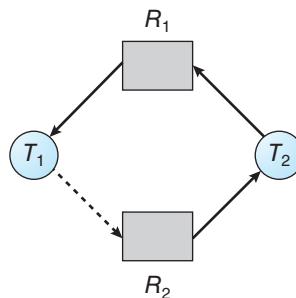


Figure 8.10 An unsafe state in a resource-allocation graph.

- **Max.** An $n \times m$ matrix defines the maximum demand of each thread. If $\text{Max}[i][j]$ equals k , then thread T_i may request at most k instances of resource type R_j .
- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each thread. If $\text{Allocation}[i][j]$ equals k , then thread T_i is currently allocated k instances of resource type R_j .
- **Need.** An $n \times m$ matrix indicates the remaining resource need of each thread. If $\text{Need}[i][j]$ equals k , then thread T_i may need k more instances of resource type R_j to complete its task. Note that $\text{Need}[i][j]$ equals $\text{Max}[i][j] - \text{Allocation}[i][j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n . We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \dots, n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as Allocation_i and Need_i . The vector Allocation_i specifies the resources currently allocated to thread T_i ; the vector Need_i specifies the additional resources that thread T_i may still request to complete its task.

8.6.3.1 Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize *Work* = *Available* and $\text{Finish}[i] = \text{false}$ for $i = 0, 1, \dots, n - 1$.
2. Find an index i such that both
 - $\text{Finish}[i] == \text{false}$
 - $\text{Need}_i \leq \text{Work}$
 If no such i exists, go to step 4.
3. $\text{Work} = \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] = \text{true}$
 Go to step 2.
4. If $\text{Finish}[i] == \text{true}$ for all i , then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

8.6.3.2 Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let Request_i be the request vector for thread T_i . If $\text{Request}_i[j] == k$, then thread T_i wants k instances of resource type R_j . When a request for resources is made by thread T_i , the following actions are taken:

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_i by modifying the state as follows:

$$\begin{aligned}\text{Available} &= \text{Available} - \text{Request}_i \\ \text{Allocation}_i &= \text{Allocation}_i + \text{Request}_i \\ \text{Need}_i &= \text{Need}_i - \text{Request}_i\end{aligned}$$

If the resulting resource-allocation state is safe, the transaction is completed, and thread T_i is allocated its resources. However, if the new state is unsafe, then T_i must wait for Request_i , and the old resource-allocation state is restored.

8.6.3.3 An Illustrative Example

To illustrate the use of the banker's algorithm, consider a system with five threads T_0 through T_4 and three resource types A , B , and C . Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that the following snapshot represents the current state of the system:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

The content of the matrix Need is defined to be $\text{Max} - \text{Allocation}$ and is as follows:

	<u>Need</u>
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence $\langle T_1, T_3, T_4, T_2, T_0 \rangle$ satisfies the safety criteria. Suppose now that thread T_1 requests one additional instance of resource type A and two instances of resource type C , so $\text{Request}_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $\text{Request}_1 \leq \text{Available}$ —that is, that

$(1,0,2) \leq (3,3,2)$, which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $\langle T_1, T_3, T_4, T_0, T_2 \rangle$ satisfies the safety requirement. Hence, we can immediately grant the request of thread T_1 .

You should be able to see, however, that when the system is in this state, a request for $(3,3,0)$ by T_4 cannot be granted, since the resources are not available. Furthermore, a request for $(0,2,0)$ by T_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm.

8.7 Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Next, we discuss these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

8.7.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from T_i to T_j in a wait-for graph implies that thread T_i is waiting for thread T_j to release a resource that T_i needs. An edge $T_i \rightarrow T_j$

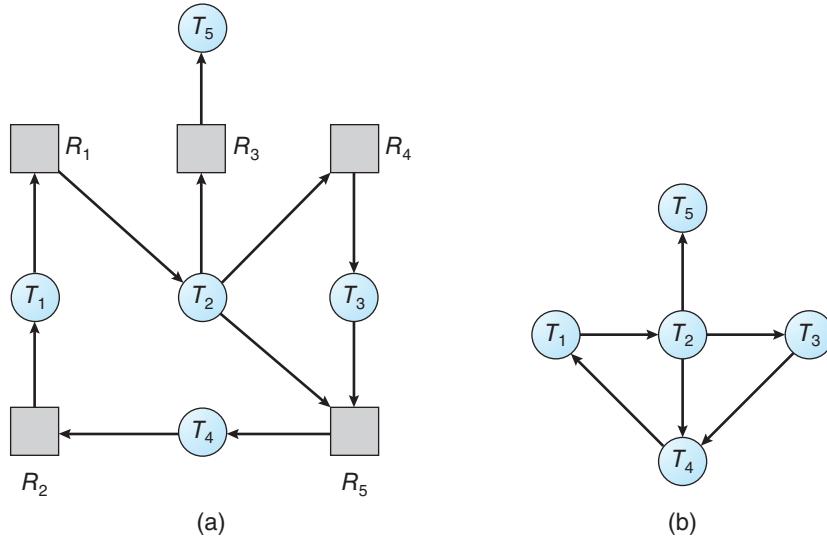


Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $T_i \rightarrow R_q$ and $R_q \rightarrow T_j$ for some resource R_q . In Figure 8.11, we present a resource-allocation graph and the corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires $O(n^2)$ operations, where n is the number of vertices in the graph.

The BCC toolkit described in Section 2.10.4 provides a tool that can detect potential deadlocks with Pthreads mutex locks in a user process running on a Linux system. The BCC tool `deadlock_detector` operates by inserting probes which trace calls to the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. When the specified process makes a call to either function, `deadlock_detector` constructs a wait-for graph of mutex locks in that process, and reports the possibility of deadlock if it detects a cycle in the graph.

8.7.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock-detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 8.6.3):

- **Available.** A vector of length m indicates the number of available resources of each type.

DEADLOCK DETECTION USING JAVA THREAD DUMPS

Although Java does not provide explicit support for deadlock detection, a **thread dump** can be used to analyze a running program to determine if there is a deadlock. A thread dump is a useful debugging tool that displays a snapshot of the states of all threads in a Java application. Java thread dumps also show locking information, including which locks a blocked thread is waiting to acquire. When a thread dump is generated, the JVM searches the wait-for graph to detect cycles, reporting any deadlocks it detects. To generate a thread dump of a running application, from the command line enter:

Ctrl-L (UNIX, Linux, or macOS)
Ctrl-Break (Windows)

In the source-code download for this text, we provide a Java example of the program shown in Figure 8.1 and describe how to generate a thread dump that reports the deadlocked Java threads.

- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each thread.
- **Request.** An $n \times m$ matrix indicates the current request of each thread. If $\text{Request}[i][j]$ equals k , then thread T_i is requesting k more instances of resource type R_j .

The \leq relation between two vectors is defined as in Section 8.6.3. To simplify notation, we again treat the rows in the matrices *Allocation* and *Request* as vectors; we refer to them as Allocation_i and Request_i . The detection algorithm described here simply investigates every possible allocation sequence for the threads that remain to be completed. Compare this algorithm with the banker's algorithm of Section 8.6.3.

1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize *Work* = *Available*. For $i = 0, 1, \dots, n-1$, if $\text{Allocation}_i \neq 0$, then $\text{Finish}[i] = \text{false}$. Otherwise, $\text{Finish}[i] = \text{true}$.

2. Find an index i such that both
 - a. $\text{Finish}[i] == \text{false}$
 - b. $\text{Request}_i \leq \text{Work}$

If no such i exists, go to step 4.

3. $\text{Work} = \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] = \text{true}$
 Go to step 2.
4. If $\text{Finish}[i] == \text{false}$ for some $i, 0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $\text{Finish}[i] == \text{false}$, then thread T_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

You may wonder why we reclaim the resources of thread T_i (in step 3) as soon as we determine that $\text{Request}_i \leq \text{Work}$ (in step 2b). We know that T_i is currently *not* involved in a deadlock (since $\text{Request}_i \leq \text{Work}$). Thus, we take an optimistic attitude and assume that T_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five threads T_0 through T_4 and three resource types A , B , and C . Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. The following snapshot represents the current state of the system:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $\langle T_0, T_2, T_3, T_1, T_4 \rangle$ results in $\text{Finish}[i] == \text{true}$ for all i .

Suppose now that thread T_2 makes one additional request for an instance of type C . The Request matrix is modified as follows:

	<u>Request</u>
	<i>A B C</i>
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by thread T_0 , the number of available resources is not sufficient to fulfill the requests of the other threads. Thus, a deadlock exists, consisting of threads T_1 , T_2 , T_3 , and T_4 .

8.7.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* threads will be affected by deadlock when it happens?

MANAGING DEADLOCK IN DATABASES

Database systems provide a useful illustration of how both open-source and commercial software manage deadlock. Updates to a database may be performed as *transactions*, and to ensure data integrity, locks are typically used. A transaction may involve several locks, so it comes as no surprise that deadlocks are possible in a database with multiple concurrent transactions. To manage deadlock, most transactional database systems include a deadlock detection and recovery mechanism. The database server will periodically search for cycles in the wait-for graph to detect deadlock among a set of transactions. When deadlock is detected, a victim is selected and the transaction is aborted and rolled back, releasing the locks held by the victim transaction and freeing the remaining transactions from deadlock. Once the remaining transactions have resumed, the aborted transaction is reissued. Choice of a victim transaction depends on the database system; for instance, MySQL attempts to select transactions that minimize the number of rows being inserted, updated, or deleted.

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked threads will be idle until the deadlock can be broken. In addition, the number of threads involved in the deadlock cycle may grow.

Deadlocks occur only when some thread makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting threads. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of threads but also the specific thread that “caused” the deadlock. (In reality, each of the deadlocked threads is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may create many cycles in the resource graph, each cycle completed by the most recent request and “caused” by the one identifiable thread.

Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and causes CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked threads “caused” the deadlock.

8.8 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system **recover** from the deadlock automatically. There

are two options for breaking a deadlock. One is simply to abort one or more threads to break the circular wait. The other is to preempt some resources from one or more of the deadlocked threads.

8.8.1 Process and Thread Termination

To eliminate deadlocks by aborting a process or thread, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it may leave that file in an incorrect state. Similarly, if the process was in the midst of updating shared data while holding a mutex lock, the system must restore the status of the lock as being available, although no guarantees can be made regarding the integrity of the shared data.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated

8.8.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of pre-emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

8.9 Summary

- Deadlock occurs in a set of processes when every process in the set is waiting for an event that can only be caused by another process in the set.
- There are four necessary conditions for deadlock: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait. Deadlock is only possible when all four conditions are present.
- Deadlocks can be modeled with resource-allocation graphs, where a cycle indicates deadlock.
- Deadlocks can be prevented by ensuring that one of the four necessary conditions for deadlock cannot occur. Of the four necessary conditions, eliminating the circular wait is the only practical approach.
- Deadlock can be avoided by using the banker's algorithm, which does not grant resources if doing so would lead the system into an unsafe state where deadlock would be possible.
- A deadlock-detection algorithm can evaluate processes and resources on a running system to determine if a set of processes is in a deadlocked state.
- If deadlock does occur, a system can attempt to recover from the deadlock by either aborting one of the processes in the circular wait or preempting resources that have been assigned to a deadlocked process.

Practice Exercises

- 8.1 List three examples of deadlocks that are not related to a computer-system environment.
- 8.2 Suppose that a system is in an unsafe state. Show that it is possible for the threads to complete their execution without entering a deadlocked state.
- 8.3 Consider the following snapshot of a system:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C D	A B C D	A B C D
T_0	0 0 1 2	0 0 1 2	1 5 2 0
T_1	1 0 0 0	1 7 5 0	
T_2	1 3 5 4	2 3 5 6	
T_3	0 6 3 2	0 6 5 2	
T_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

- a. What is the content of the matrix *Need*?
- b. Is the system in a safe state?
- c. If a request from thread T_1 arrives for $(0,4,2,0)$, can the request be granted immediately?
- 8.4 A possible method for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects $A \dots E$, deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent deadlock by adding a sixth object F . Whenever a thread wants to acquire the synchronization lock for any object $A \dots E$, it must first acquire the lock for object F . This solution is known as **containment**: the locks for objects $A \dots E$ are contained within the lock for object F . Compare this scheme with the circular-wait scheme of Section 8.5.4.
- 8.5 Prove that the safety algorithm presented in Section 8.6.3 requires an order of $m \times n^2$ operations.
- 8.6 Consider a computer system that runs 5,000 jobs per month and has no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about ten jobs per deadlock. Each job is worth about two dollars (in CPU time), and the jobs terminated tend to be about half done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase of about 10 percent in the average execution time per

job. Since the machine currently has 30 percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

- a. What are the arguments for installing the deadlock-avoidance algorithm?
 - b. What are the arguments against installing the deadlock-avoidance algorithm?
- 8.7 Can a system detect that some of its threads are starving? If you answer “yes,” explain how it can. If you answer “no,” explain how the system can deal with the starvation problem.
- 8.8 Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any threads that are blocked waiting for resources. If a blocked thread has the desired resources, then these resources are taken away from it and are given to the requesting thread. The vector of resources for which the blocked thread is waiting is increased to include the resources that were taken away.
- For example, a system has three resource types, and the vector *Available* is initialized to (4,2,2). If thread T_0 asks for (2,2,1), it gets them. If T_1 asks for (1,0,1), it gets them. Then, if T_0 asks for (0,0,1), it is blocked (resource not available). If T_2 now asks for (2,0,0), it gets the available one (1,0,0), as well as one that was allocated to T_0 (since T_0 is blocked). T_0 ’s *Allocation* vector goes down to (1,2,1), and its *Need* vector goes up to (1,0,1).
- a. Can deadlock occur? If you answer “yes,” give an example. If you answer “no,” specify which necessary condition cannot occur.
 - b. Can indefinite blocking occur? Explain your answer.
- 8.9 Consider the following snapshot of a system:

	<i>Allocation</i>	<i>Max</i>
	A B C D	A B C D
T_0	3 0 1 4	5 1 1 7
T_1	2 2 1 0	3 2 1 1
T_2	3 1 2 1	3 3 2 1
T_3	0 5 1 0	4 6 1 2
T_4	4 2 1 2	6 3 2 5

Using the banker’s algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

- a. $\text{Available} = (0, 3, 0, 1)$
- b. $\text{Available} = (1, 0, 0, 2)$

- 8.10 Suppose that you have coded the deadlock-avoidance safety algorithm that determines if a system is in a safe state or not, and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining $\text{Max}_i = \text{Waiting}_i + \text{Allocation}_i$, where Waiting_i is a vector specifying the resources for which thread i is waiting and Allocation_i is as defined in Section 8.6? Explain your answer.
- 8.11 Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

Further Reading

Most research involving deadlock was conducted many years ago. [Dijkstra (1965)] was one of the first and most influential contributors in the deadlock area.

Details of how the MySQL database manages deadlock can be found at <http://dev.mysql.com/>.

Details on the lockdep tool can be found at <https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt>.

Bibliography

[Dijkstra (1965)] E. W. Dijkstra, “Cooperating Sequential Processes”, Technical report, Technological University, Eindhoven, the Netherlands (1965).

Chapter 8 Exercises

- 8.12 Consider the traffic deadlock depicted in Figure 8.12.
- Show that the four necessary conditions for deadlock hold in this example.
 - State a simple rule for avoiding deadlocks in this system.
- 8.13 Draw the resource-allocation graph that illustrates deadlock from the program example shown in Figure 8.1 in Section 8.2.
- 8.14 In Section 6.8.1, we described a potential deadlock scenario involving processes P_0 and P_1 and semaphores S and Q. Draw the resource-allocation graph that illustrates deadlock under the scenario presented in that section.
- 8.15 Assume that a multithreaded application uses only reader–writer locks for synchronization. Applying the four necessary conditions for deadlock, is deadlock still possible if multiple reader–writer locks are used?
- 8.16 The program example shown in Figure 8.1 doesn't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.
- 8.17 In Section 8.5.4, we described a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if two threads simultaneously invoke the `transaction()` function. Fix the `transaction()` function to prevent deadlocks.

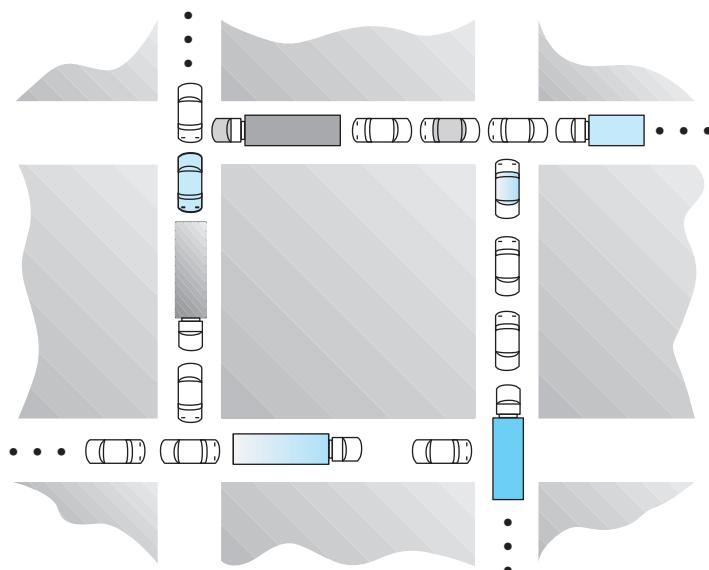


Figure 8.11 Traffic deadlock for Exercise 8.12.

- 8.18** Which of the six resource-allocation graphs shown in Figure 8.12 illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.
- 8.19** Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:
- Runtime overhead
 - System throughput
- 8.20** In a real computer system, neither the resources available nor the demands of threads for resources are consistent over long periods (months). Resources break or are replaced, new processes and threads come and go, and new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the

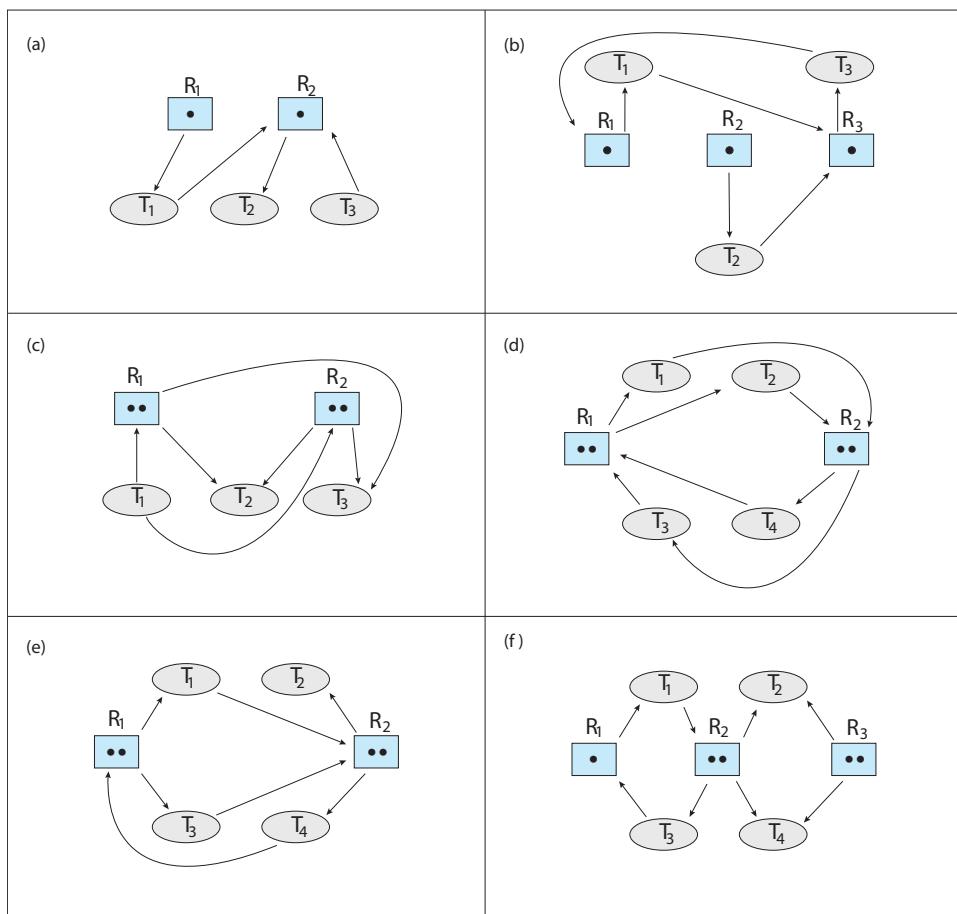


Figure 8.12 Resource-allocation graphs for Exercise 8.18.

following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

- Increase *Available* (new resources added).
- Decrease *Available* (resource permanently removed from system).
- Increase *Max* for one thread (the thread needs or wants more resources than allowed).
- Decrease *Max* for one thread (the thread decides it does not need that many resources).
- Increase the number of threads.
- Decrease the number of threads.

8.21 Consider the following snapshot of a system:

	<u>Allocation</u>				<u>Max</u>			
	A	B	C	D	A	B	C	D
T_0	2	1	0	6	6	3	2	7
T_1	3	3	1	3	5	4	1	5
T_2	2	3	1	2	6	6	1	4
T_3	1	2	3	4	4	3	4	5
T_4	3	0	3	0	7	2	6	1

What are the contents of the *Need* matrix?

- 8.22** Consider a system consisting of four resources of the same type that are shared by three threads, each of which needs at most two resources. Show that the system is deadlock free.
- 8.23** Consider a system consisting of m resources of the same type being shared by n threads. A thread can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:
- The maximum need of each thread is between one resource and m resources.
 - The sum of all maximum needs is less than $m + n$.
- 8.24** Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.
- 8.25** Consider again the setting in the preceding exercise. Assume now that each philosopher requires three chopsticks to eat. Resource requests are still issued one at a time. Describe some simple rules for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

- 8.26** We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1.

Show through an example that we cannot implement the multiple-resource-type banker's scheme by applying the single-resource-type scheme to each resource type individually.

- 8.27** Consider the following snapshot of a system:

	<u>Allocation</u>				<u>Max</u>			
	A	B	C	D	A	B	C	D
T_0	1	2	0	2	4	3	1	6
T_1	0	1	1	2	2	4	2	4
T_2	1	2	4	0	3	6	5	1
T_3	1	2	0	1	2	6	2	3
T_4	1	0	0	1	3	1	1	2

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

- a. $\text{Available} = (2, 2, 2, 3)$
- b. $\text{Available} = (4, 4, 1, 1)$
- c. $\text{Available} = (3, 0, 1, 4)$
- d. $\text{Available} = (1, 5, 2, 2)$

- 8.28** Consider the following snapshot of a system:

	<u>Allocation</u>				<u>Max</u>				<u>Available</u>			
	A	B	C	D	A	B	C	D	A	B	C	D
T_0	3	1	4	1	6	4	7	3	2	2	2	4
T_1	2	1	0	2	4	2	3	2				
T_2	2	4	1	3	2	5	3	3				
T_3	4	1	1	0	6	3	3	2				
T_4	2	2	2	1	5	6	7	5				

Answer the following questions using the banker's algorithm:

- a. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.
- b. If a request from thread T_4 arrives for $(2, 2, 2, 4)$, can the request be granted immediately?
- c. If a request from thread T_2 arrives for $(0, 1, 1, 0)$, can the request be granted immediately?
- d. If a request from thread T_3 arrives for $(2, 2, 1, 2)$, can the request be granted immediately?

- 8.29** What is the optimistic assumption made in the deadlock-detection algorithm? How can this assumption be violated?
- 8.30** A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if a northbound and a southbound farmer get on the bridge at the same time. (Vermont farmers are stubborn and are unable to back up.) Using semaphores and/or mutex locks, design an algorithm in pseudocode that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).
- 8.31** Modify your solution to Exercise 8.30 so that it is starvation-free.

Programming Problems

- 8.32** Implement your solution to Exercise 8.30 using POSIX synchronization. In particular, represent northbound and southbound farmers as separate threads. Once a farmer is on the bridge, the associated thread will sleep for a random period of time, representing traveling across the bridge. Design your program so that you can create several threads representing the northbound and southbound farmers.
- 8.33** In Figure 8.7, we illustrate a `transaction()` function that dynamically acquires locks. In the text, we describe how this function presents difficulties for acquiring locks in a way that avoids deadlock. Using the Java implementation of `transaction()` that is provided in the source-code download for this text, modify it using the `System.identityHashCode()` method so that the locks are acquired in order.

Programming Projects

Banker's Algorithm

For this project, you will write a program that implements the banker's algorithm discussed in Section 8.6.3. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. Although the code examples that describe this project are illustrated in C, you may also develop a solution using Java.

The Banker

The banker will consider requests from n customers for m resources types, as outlined in Section 8.6.3. The banker will keep track of the resources using the following data structures:

```
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 4

/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES] ;

/*the maximum demand of each customer */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES] ;

/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES] ;

/* the remaining need of each customer */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES] ;
```

The banker will grant a request if it satisfies the safety algorithm outlined in Section 8.6.3.1. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows:

```
int request_resources(int customer_num, int request[]);
void release_resources(int customer_num, int release[]);
```

The `request_resources()` function should return 0 if successful and -1 if unsuccessful.

Testing Your Implementation

Design a program that allows the user to interactively enter a request for resources, to release resources, or to output the values of the different data structures (`available`, `maximum`, `allocation`, and `need`) used with the banker's algorithm.

You should invoke your program by passing the number of resources of each type on the command line. For example, if there were four resource types, with ten instances of the first type, five of the second type, seven of the third type, and eight of the fourth type, you would invoke your program as follows:

```
./a.out 10 5 7 8
```

The `available` array would be initialized to these values.

Your program will initially read in a file containing the maximum number of requests for each customer. For example, if there are five customers and four resources, the input file would appear as follows:

```
6,4,7,3
4,2,3,2
2,5,3,3
6,3,3,2
5,6,7,5
```

where each line in the input file represents the maximum request of each resource type for each customer. Your program will initialize the `maximum` array to these values.

Your program will then have the user enter commands responding to a request of resources, a release of resources, or the current values of the different data structures. Use the command 'RQ' for requesting resources, 'RL' for releasing resources, and '*' to output the values of the different data structures. For example, if customer 0 were to request the resources (3,1,2,1), the following command would be entered:

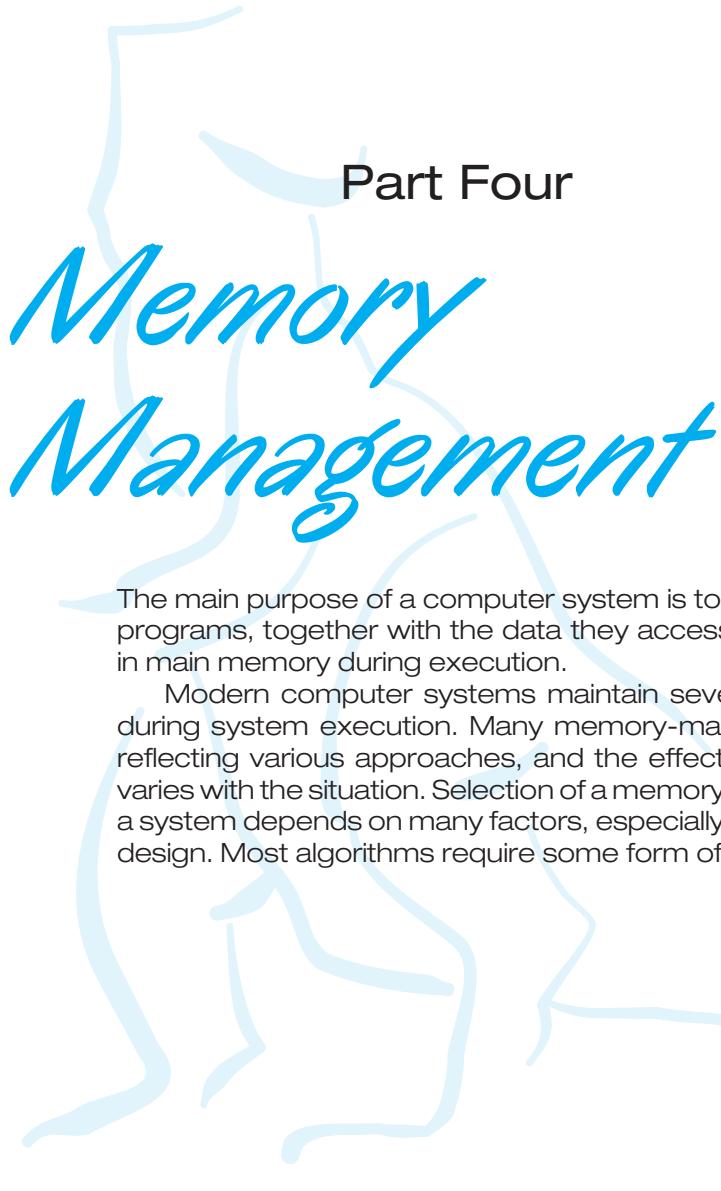
```
RQ 0 3 1 2 1
```

Your program would then output whether the request would be satisfied or denied using the safety algorithm outlined in Section 8.6.3.1.

Similarly, if customer 4 were to release the resources (1, 2, 3, 1), the user would enter the following command:

RL 4 1 2 3 1

Finally, if the command '*' is entered, your program would output the values of the available, maximum, allocation, and need arrays.



Part Four

Memory Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution.

Modern computer systems maintain several processes in memory during system execution. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm varies with the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the system's *hardware* design. Most algorithms require some form of hardware support.

Main Memory



In Chapter 5, we showed how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep many processes in memory—that is, we must share memory.

In this chapter, we discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to a strategy that uses paging. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system. As we shall see, most algorithms require hardware support, leading many systems to have closely integrated hardware and operating-system memory management.

CHAPTER OBJECTIVES

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses.
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain the distinction between internal and external fragmentation.
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB).
- Describe hierarchical paging, hashed paging, and inverted page tables.
- Describe address translation for IA-32, x86-64, and ARMv8 architectures.

9.1 Background

As we saw in Chapter 1, memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of

the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

We begin our discussion by covering several issues that are pertinent to managing memory: basic hardware, the binding of symbolic (or virtual) memory addresses to actual physical addresses, and the distinction between logical and physical addresses. We conclude the section with a discussion of dynamic linking and shared libraries.

9.1.1 Basic Hardware

Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into each CPU core are generally accessible within one cycle of the CPU clock. Some CPU cores can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. Such a **cache** was described in Section 1.5.5. To manage a cache built into the CPU, the hardware automatically speeds up memory access without any operating-system control. (Recall from Section 5.5.2 that during a memory stall, a multithreaded core can switch from the stalled hardware thread to another hardware thread.)

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation, we must protect the operating system from access by user processes, as well as protect user processes from one another. This protection must be provided by the hardware, because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). Hardware implements this protection in several different ways, as we show throughout the chapter. Here, we outline one possible implementation.

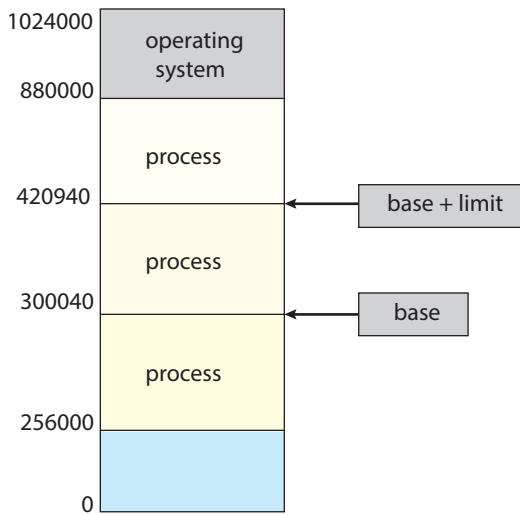


Figure 9.1 A base and a limit register define a logical address space.

We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 9.1. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 9.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a

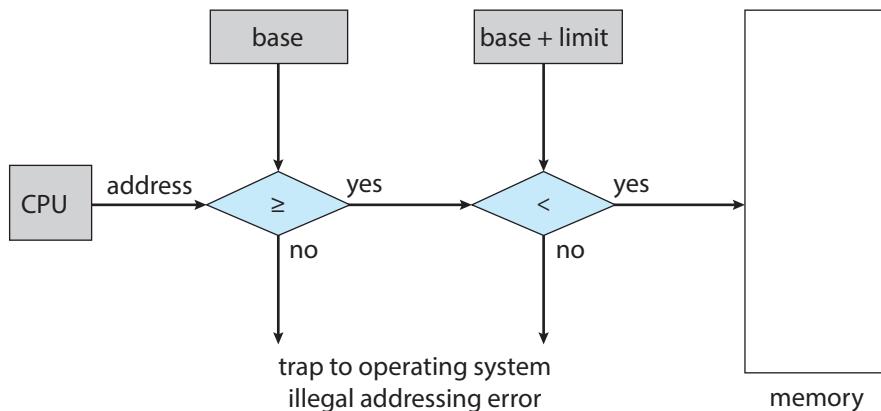


Figure 9.2 Hardware address protection with base and limit registers.

multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

9.1.2 Address Binding

Usually, a program resides on a disk as a binary executable file. To run, the program must be brought into memory and placed within the context of a process (as described in Section 2.5), where it becomes eligible for execution on an available CPU. As the process executes, it accesses instructions and data from memory. Eventually, the process terminates, and its memory is reclaimed for use by other processes.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. You will see later how the operating system actually places a process in physical memory.

In most cases, a user program goes through several steps—some of which may be optional—before being executed (Figure 9.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically **binds** these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”). The linker or loader (see Section 2.5) in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time.** If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location R , then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

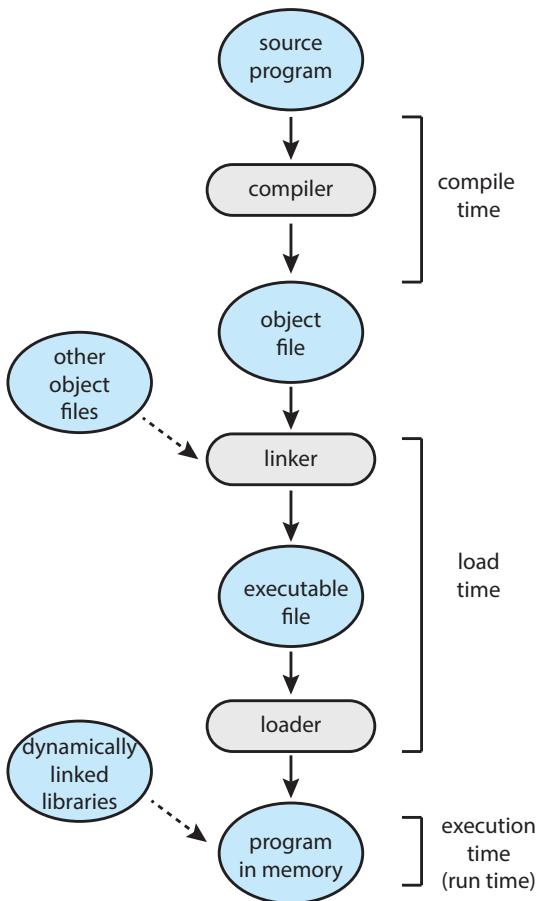


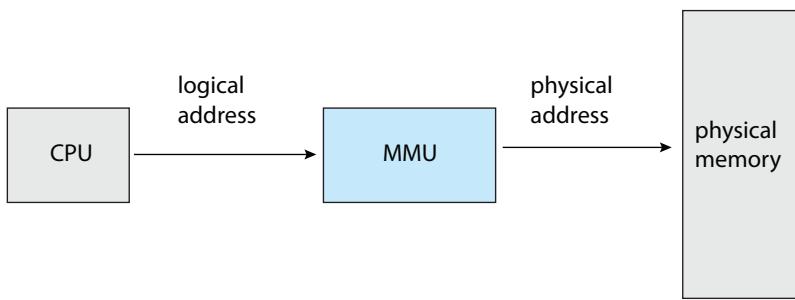
Figure 9.3 Multistep processing of a user program.

- **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 9.1.3. Most operating systems use this method.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

9.1.3 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into

**Figure 9.4** Memory management unit (MMU).

the **memory-address register** of the memory—is commonly referred to as a **physical address**.

Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use *logical address* and *virtual address* interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)** (Figure 9.4). We can choose from many different methods to accomplish such mapping, as we discuss in Section 9.2 through Section 9.3. For the time being, we illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme described in Section 9.1.1. The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 9.5). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

The user program never accesses the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 9.1.2. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range *R* + 0 to *R* + *max* for a base value *R*). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used. The

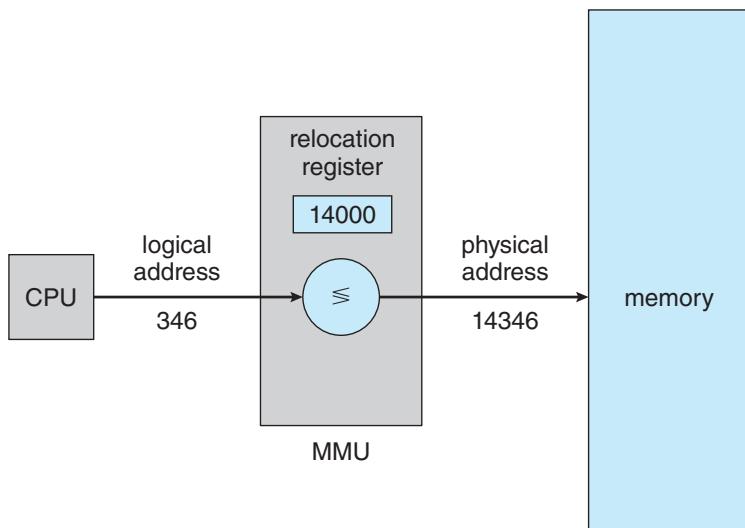


Figure 9.5 Dynamic relocation using a relocation register.

concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

9.1.4 Dynamic Loading

In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In such a situation, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

9.1.5 Dynamic Linking and Shared Libraries

Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run (refer back to Figure 9.3). Some operating systems support only **static linking**, in which system libraries are treated

like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as the standard C language library. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement not only increases the size of an executable image but also may waste main memory. A second advantage of DLLs is that these libraries can be shared among multiple processes, so that only one instance of the DLL in main memory. For this reason, DLLs are also known as **shared libraries**, and are used extensively in Windows and Linux systems.

When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary. It then adjusts addresses that reference functions in the dynamic library to the location in memory where the DLL is stored.

Dynamically linked libraries can be extended to library updates (such as bug fixes). In addition, a library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number. Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library.

Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses. We elaborate on this concept, as well as how DLLs can be shared by multiple processes, when we discuss paging in Section 9.3.4.

9.2 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the operating system and one for the user processes. We can place the operating system in either low memory addresses or high memory addresses. This decision depends on many factors, such as the location of the interrupt vector. However, many operating systems (including Linux and Windows) place the operating system in high memory, and therefore we discuss only that situation.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process. Before discussing this memory allocation scheme further, though, we must address the issue of memory protection.

9.2.1 Memory Protection

We can prevent a process from accessing memory that it does not own by combining two ideas previously discussed. If we have a system with a relocation register (Section 9.1.3), together with a limit register (Section 9.1.1), we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 9.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver is not currently in use, it makes little sense to keep it in memory; instead, it can be loaded into memory only when it is needed. Likewise, when the device driver is no longer needed, it can be removed and its memory allocated for other needs.

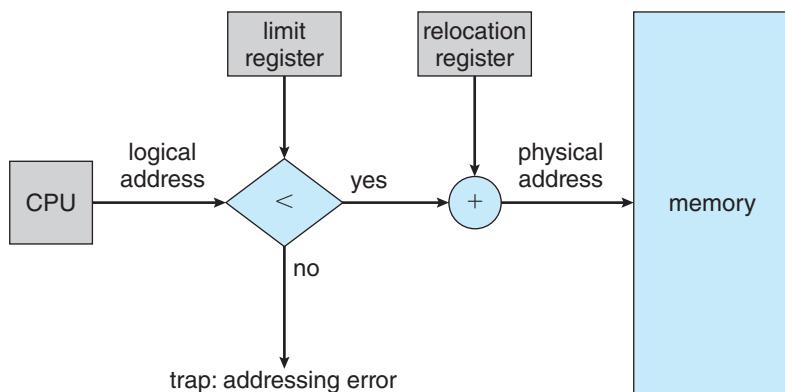


Figure 9.6 Hardware support for relocation and limit registers.

9.2.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process. In this **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, as you will see, memory contains a set of holes of various sizes.

Figure 9.7 depicts this scheme. Initially, the memory is fully utilized, containing processes 5, 8, and 2. After process 8 leaves, there is one contiguous hole. Later on, process 9 arrives and is allocated memory. Then process 5 departs, resulting in two noncontiguous holes.

As processes enter the system, the operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process.

What happens when there isn't sufficient memory to satisfy the demands of an arriving process? One option is to simply reject the process and provide an appropriate error message. Alternatively, we can place such processes into a wait queue. When memory is later released, the operating system checks the wait queue to determine if it will satisfy the memory demands of a waiting process.

In general, as mentioned, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general **dynamic storage-allocation problem**, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

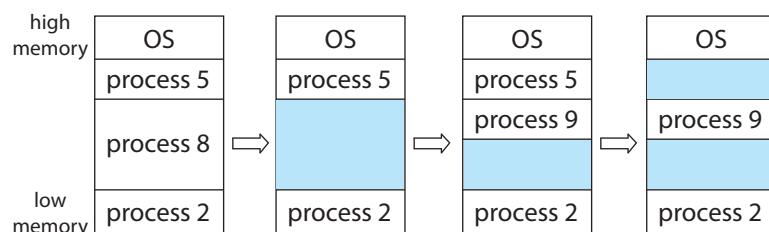


Figure 9.7 Variable partition.

- **First fit.** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit.** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit.** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

9.2.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5 N$ blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. This is the strategy used in **paging**, the most common memory-management technique for computer systems. We describe paging in the following section.

Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data. We discuss the topic further in the storage management chapters (Chapter 11 through Chapter 15).

9.3 Paging

Memory management discussed thus far has required the physical address space of a process to be contiguous. We now introduce **paging**, a memory-management scheme that permits a process's physical address space to be non-contiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation. Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices. Paging is implemented through cooperation between the operating system and the computer hardware.

9.3.1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**:

page number	page offset
p	d

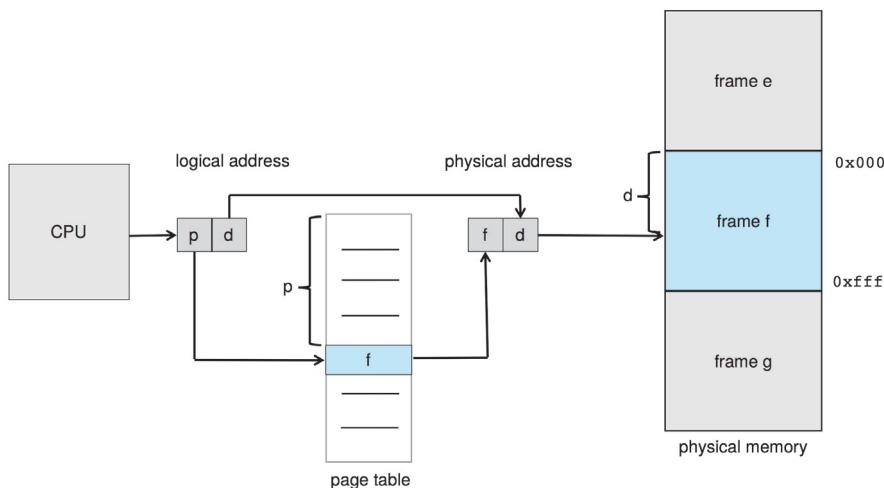


Figure 9.8 Paging hardware.

The page number is used as an index into a per-process **page table**. This is illustrated in Figure 9.8. The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address. The paging model of memory is shown in Figure 9.9.

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.
2. Extract the corresponding frame number f from the page table.
3. Replace the page number p in the logical address with the frame number f .

As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2^m , and a page size is 2^n bytes, then the high-order $m-n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number	page offset
p	d
$m-n$	n

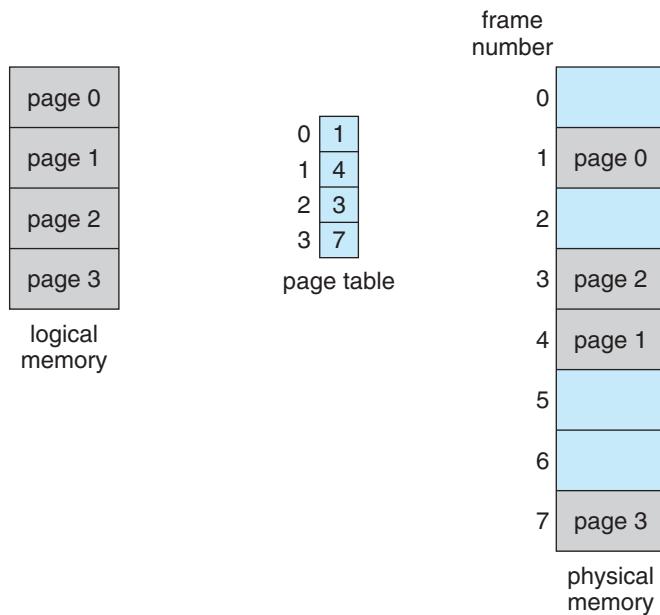


Figure 9.9 Paging model of logical and physical memory.

where p is an index into the page table and d is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure 9.10. Here, in the logical address, $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= $(5 \times 4) + 0$]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= $(5 \times 4) + 3$]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= $(6 \times 4) + 0$]. Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of $2,048 - 1,086 = 962$ bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated $n + 1$ frames, resulting in internal fragmentation of almost an entire frame.

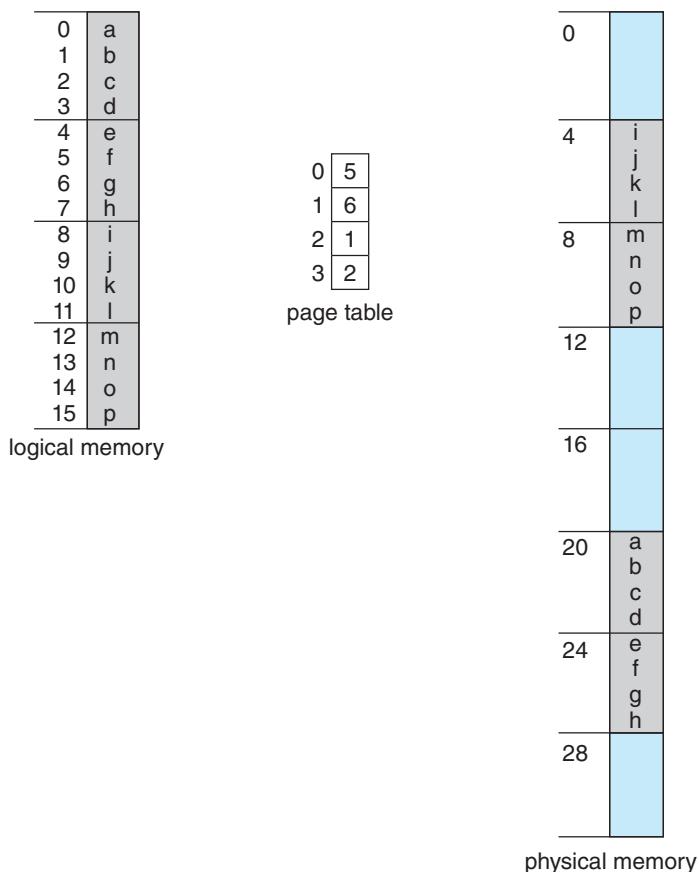


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount of data being transferred is larger (Chapter 11). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages are typically either 4 KB or 8 KB in size, and some systems support even larger page sizes. Some CPUs and operating systems even support multiple page sizes. For instance, on x86-64 systems, Windows 10 supports page sizes of 4 KB and 2 MB. Linux also supports two page sizes: a default page size (typically 4 KB) and an architecture-dependent larger page size called **huge pages**.

Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames. If the frame size is 4 KB (2^{12}), then a system with 4-byte entries can address 2^{44} bytes (or 16 TB) of physical memory. We should note here that the size of physical memory in a paged memory system is typically different from the maximum logical size of a process. As we further explore paging, we will

OBTAINING THE PAGE SIZE ON LINUX SYSTEMS

On a Linux system, the page size varies according to architecture, and there are several ways of obtaining the page size. One approach is to use the system call `getpagesize()`. Another strategy is to enter the following command on the command line:

```
getconf PAGESIZE
```

Each of these techniques returns the page size as a number of bytes.

introduce other information that must be kept in the page-table entries. That information reduces the number of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 9.11).

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds

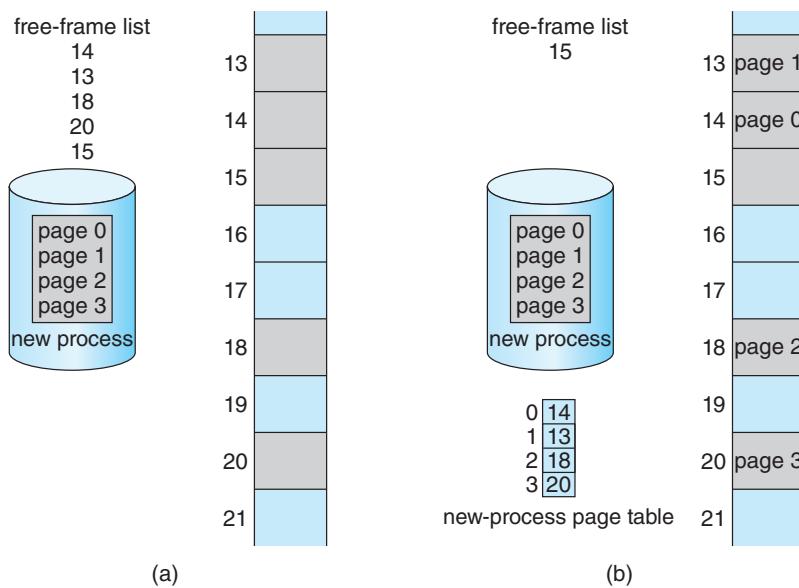


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

other programs. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a single, system-wide data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

9.3.2 Hardware Support

As page tables are per-process data structures, a pointer to the page table is stored with the other register values (like the instruction pointer) in the process control block of each process. When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient. However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary CPUs, however, support much larger page tables (for example, 2^{20} entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

9.3.2.1 Translation Look-Aside Buffer

Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times. Suppose we want to access location i . We must first index into the page table, using the value in

the PTBR offset by the page number for i . This task requires one memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, *two* memory accesses are needed to access data (one for the page-table entry and one for the actual data). Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances.

The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), address translation proceeds following the steps illustrated in Section 9.3.1, where a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 9.12). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifier (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the

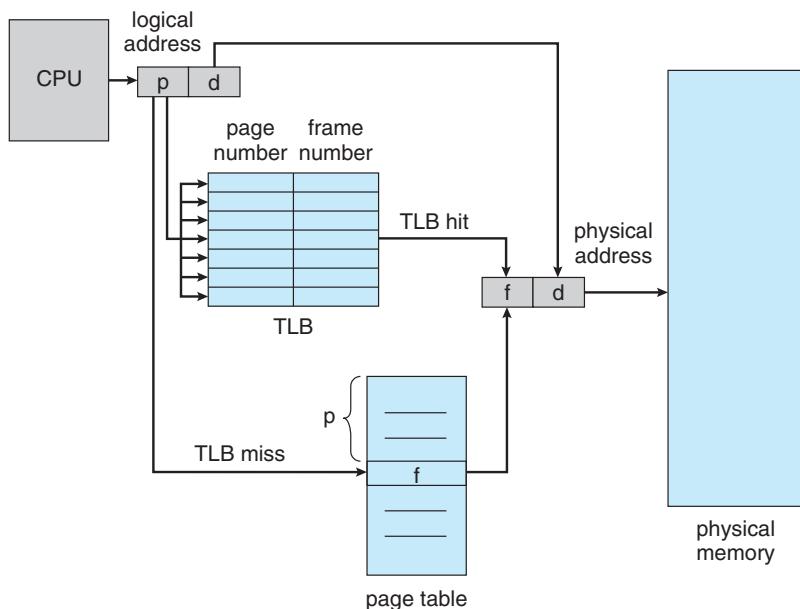


Figure 9.12 Paging hardware with TLB.

wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 10 + 0.20 \times 20 \\ &= 12 \text{ nanoseconds}\end{aligned}$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 10 to 12 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have

$$\begin{aligned}\text{effective access time} &= 0.99 \times 10 + 0.01 \times 20 \\ &= 10.1 \text{ nanoseconds}\end{aligned}$$

This increased hit rate produces only a 1 percent slowdown in access time.

As noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work.

A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs). We will further explore the impact of the hit ratio on the TLB in Chapter 10.

TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between different generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

9.3.3 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to *valid*, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to *invalid*, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure 9.13. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the

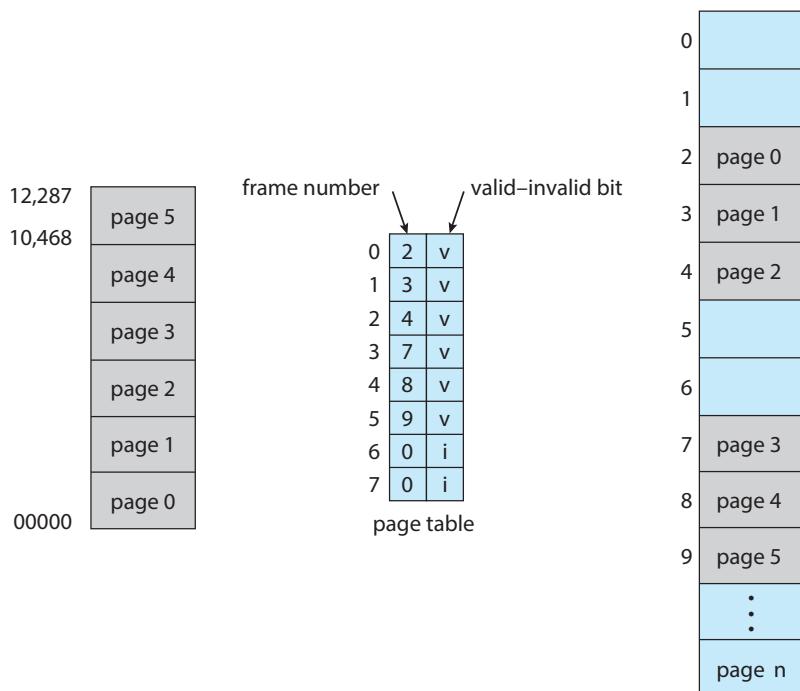


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register (PTLR)**, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

9.3.4 Shared Pages

An advantage of paging is the possibility of *sharing* common code, a consideration that is particularly important in an environment with multiple processes. Consider the standard C library, which provides a portion of the system call interface for many versions of UNIX and Linux. On a typical Linux system, most user processes require the standard C library `libc`. One option is to have

each process load its own copy of `libc` into its address space. If a system has 40 user processes, and the `libc` library is 2 MB, this would require 80 MB of memory.

If the code is **reentrant code**, however, it can be shared, as shown in Figure 9.14. Here, we see three processes sharing the pages for the standard C library `libc`. (Although the figure shows the `libc` library occupying four pages, in reality, it would occupy more.) Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the standard C library need be kept in physical memory, and the page table for each user process maps onto the same physical copy of `libc`. Thus, to support 40 processes, we need only one copy of the library, and the total space now required is 2 MB instead of 80 MB—a significant saving!

In addition to run-time libraries such as `libc`, other heavily used programs can also be shared—compilers, window systems, database systems, and so on. The shared libraries discussed in Section 9.1.5 are typically implemented with shared pages. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.

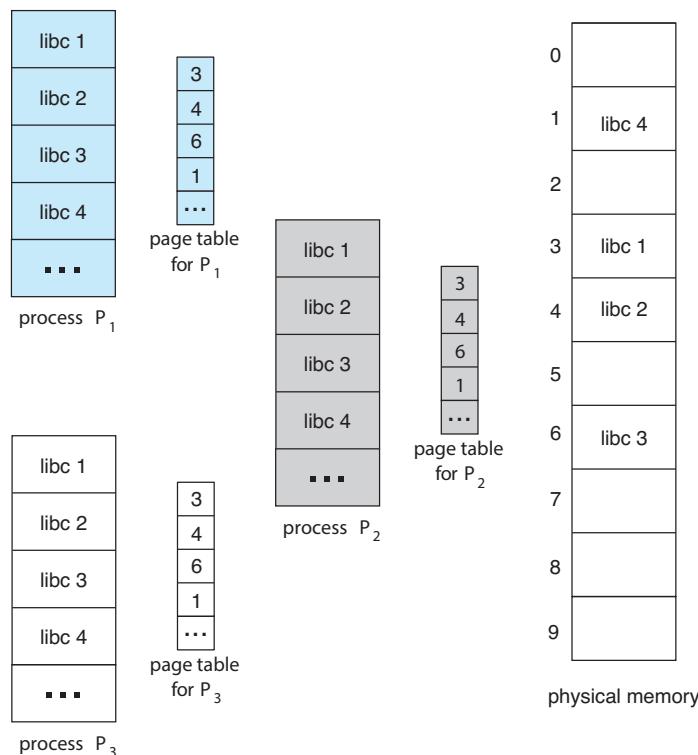


Figure 9.14 Sharing of standard C library in a paging environment.

The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads, described in Chapter 4. Furthermore, recall that in Chapter 3 we described shared memory as a method of interprocess communication. Some operating systems implement shared memory using shared pages.

Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages. We cover several other benefits in Chapter 10.

9.4 Structure of the Page Table

In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

9.4.1 Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of over 1 million entries ($2^{20} = 2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 9.15). For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table and p_2 is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure 9.16. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped** page table.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let's suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:

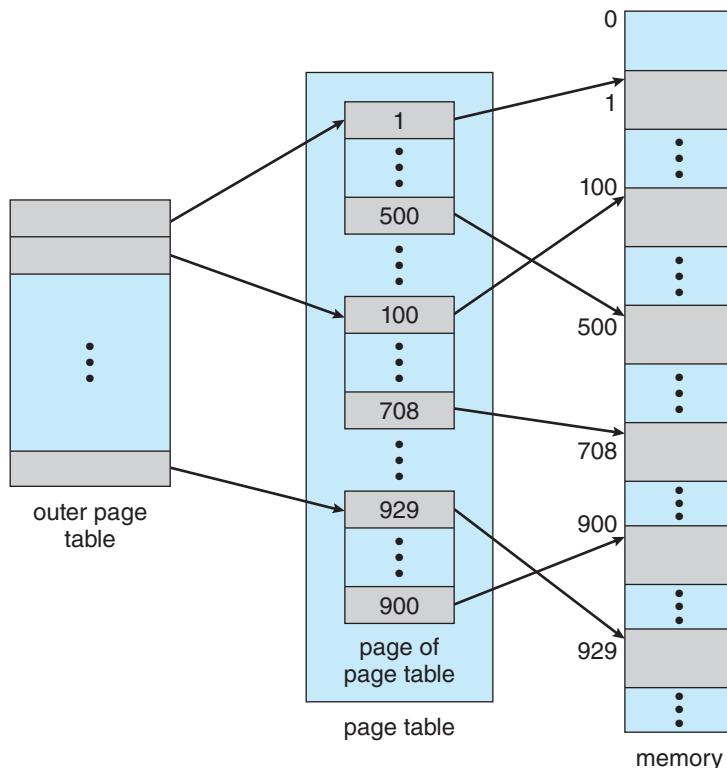


Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
p_1	p_2	d

42 10 12

The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. (This approach is also used on some 32-bit processors for added flexibility and efficiency.)

We can divide the outer page table in various ways. For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes). In this case, a 64-bit address space is still daunting:

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d

32 10 10 12

The outer page table is still 2^{34} bytes (16 GB) in size.

The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth. The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—

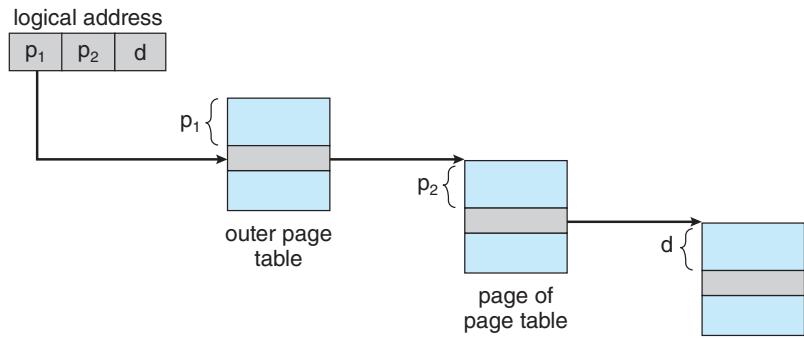


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

to translate each logical address. You can see from this example why, for 64-bit architectures, hierarchical page tables are generally considered inappropriate.

9.4.2 Hashed Page Tables

One approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 9.17.

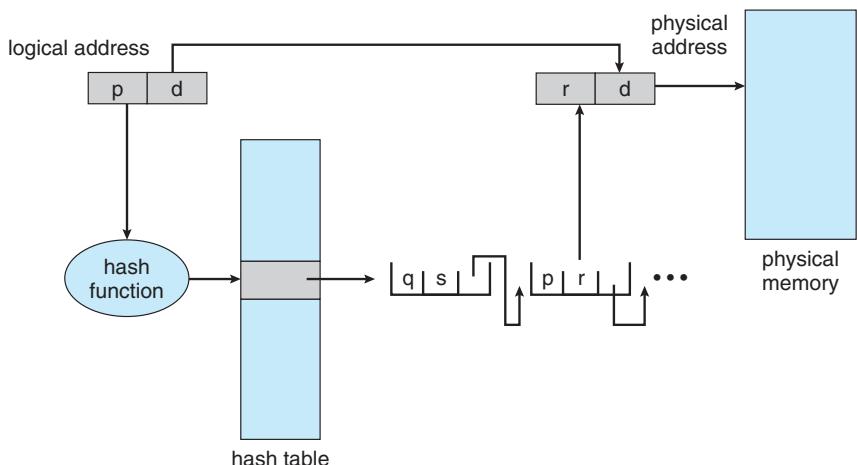


Figure 9.17 Hashed page table.

A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

9.4.3 Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 9.18 shows the operation of an inverted page table. Compare it with Figure 9.8, which depicts a standard page table in operation. Inverted page tables often require that an address-space identifier (Section 9.3.2) be stored in each entry of the page table, since the table usually contains several

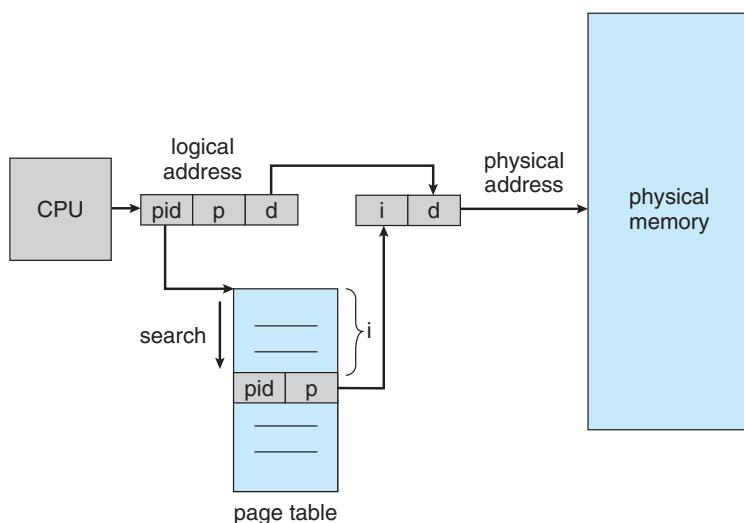


Figure 9.18 Inverted page table.

different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, we describe a simplified version of the inverted page table used in the IBM RT. IBM was the first major company to use inverted page tables, starting with the IBM System 38 and continuing through the RS/6000 and the current IBM Power CPUs. For the IBM RT, each virtual address in the system consists of a triple:

$$\langle \text{process-id}, \text{page-number}, \text{offset} \rangle.$$

Each inverted page-table entry is a pair $\langle \text{process-id}, \text{page-number} \rangle$ where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of $\langle \text{process-id}, \text{page-number} \rangle$, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry i —then the physical address $\langle i, \text{offset} \rangle$ is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long. To alleviate this problem, we use a hash table, as described in Section 9.4.2, to limit the search to one—or at most a few—page-table entries. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table. (Recall that the TLB is searched first, before the hash table is consulted, offering some performance improvement.)

One interesting issue with inverted page tables involves shared memory. With standard paging, each process has its own page table, which allows multiple virtual addresses to be mapped to the same physical address. This method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. Therefore, with inverted page tables, only one mapping of a virtual address to the shared physical address may occur at any given time. A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address.

9.4.4 Oracle SPARC Solaris

Consider as a final example a modern 64-bit CPU and operating system that are tightly integrated to provide low-overhead virtual memory. **Solaris** running on the **SPARC** CPU is a fully 64-bit operating system and as such has to solve the problem of virtual memory without using up all of its physical memory by keeping multiple levels of page tables. Its approach is a bit complex but solves the problem efficiently using hashed page tables. There are two hash tables—one for the kernel and one for all user processes. Each maps memory addresses from virtual to physical memory. Each hash-table entry represents a contiguous area of mapped virtual memory, which is more efficient than

having a separate hash-table entry for each page. Each entry has a base address and a span indicating the number of pages the entry represents.

Virtual-to-physical translation would take too long if each address required searching through a hash table, so the CPU implements a TLB that holds translation table entries (TTEs) for fast hardware lookups. A cache of these TTEs resides in a translation storage buffer (TSB), which includes an entry per recently accessed page. When a virtual address reference occurs, the hardware searches the TLB for a translation. If none is found, the hardware walks through the in-memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup. This **TLB walk** functionality is found on many modern CPUs. If a match is found in the TSB, the CPU copies the TSB entry into the TLB, and the memory translation completes. If no match is found in the TSB, the kernel is interrupted to search the hash table. The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit. Finally, the interrupt handler returns control to the MMU, which completes the address translation and retrieves the requested byte or word from main memory.

9.5 Swapping

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 9.19). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

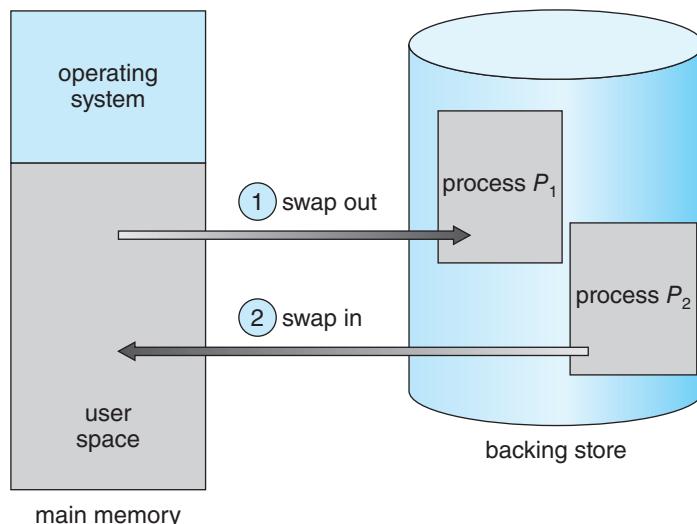


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

9.5.1 Standard Swapping

Standard swapping involves moving entire processes between main memory and a backing store. The backing store is commonly fast secondary storage. It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images. When a process or part is swapped to the backing store, the data structures associated with the process must be written to the backing store. For a multithreaded process, all per-thread data structures must be swapped as well. The operating system must also maintain metadata for processes that have been swapped out, so they can be restored when they are swapped back in to memory.

The advantage of standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them. Idle or mostly idle processes are good candidates for swapping; any memory that has been allocated to these inactive processes can then be dedicated to active processes. If an inactive process that has been swapped out becomes active once again, it must then be swapped back in. This is illustrated in Figure 9.19.

9.5.2 Swapping with Paging

Standard swapping was used in traditional UNIX systems, but it is generally no longer used in contemporary operating systems, because the amount of time required to move entire processes between memory and the backing store is prohibitive. (An exception to this is Solaris, which still uses standard swapping, however only under dire circumstances when available memory is extremely low.)

Most systems, including Linux and Windows, now use a variation of swapping in which pages of a process—rather than an entire process—can be swapped. This strategy still allows physical memory to be oversubscribed, but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping. In fact, the term *swapping* now generally refers to standard swapping, and *paging* refers to swapping with paging. A **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**. Swapping with paging is illustrated in Figure 9.20 where a subset of pages for processes A and B are being paged-out and paged-in respectively. As we shall see in Chapter 10, swapping with paging works well in conjunction with virtual memory.

9.5.3 Swapping on Mobile Systems

Most operating systems for PCs and servers support swapping pages. In contrast, mobile systems typically do not support swapping in any form. Mobile devices generally use flash memory rather than more spacious hard disks for nonvolatile storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

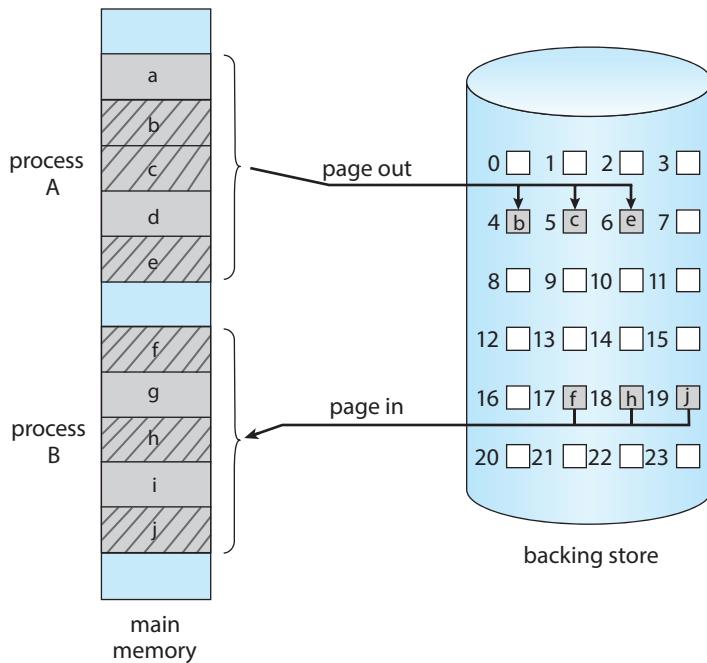


Figure 9.20 Swapping with paging.

Instead of using swapping, when free memory falls below a certain threshold, Apple's iOS *asks* applications to voluntarily relinquish allocated memory. Read-only data (such as code) are removed from main memory and later reloaded from flash memory if necessary. Data that have been modified (such as the stack) are never removed. However, any applications that fail to free up sufficient memory may be terminated by the operating system.

Android adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its **application state** to flash memory so that it can be quickly restarted.

Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or suffer from memory leaks.

SYSTEM PERFORMANCE UNDER SWAPPING

Although swapping pages is more efficient than swapping entire processes, when a system is undergoing *any* form of swapping, it is often a sign there are more active processes than available physical memory. There are generally two approaches for handling this situation: (1) terminate some processes, or (2) get more physical memory!

9.6 Example: Intel 32- and 64-bit Architectures

The architecture of Intel chips has dominated the personal computer landscape for decades. The 16-bit Intel 8086 appeared in the late 1970s and was soon followed by another 16-bit chip—the Intel 8088—which was notable for being the chip used in the original IBM PC. Intel later produced a series of 32-bit chips—the IA-32—which included the family of 32-bit Pentium processors. More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture. Currently, all the most popular PC operating systems run on Intel chips, including Windows, macOS, and Linux (although Linux, of course, runs on several other architectures as well). Notably, however, Intel’s dominance has not spread to mobile systems, where the ARM architecture currently enjoys considerable success (see Section 9.7).

In this section, we examine address translation for both IA-32 and x86-64 architectures. Before we proceed, however, it is important to note that because Intel has released several versions—as well as variations—of its architectures over the years, we cannot provide a complete description of the memory-management structure of all its chips. Nor can we provide all of the CPU details, as that information is best left to books on computer architecture. Rather, we present the major memory-management concepts of these Intel CPUs.

9.6.1 IA-32 Architecture

Memory management in IA-32 systems is divided into two components—segmentation and paging—and works as follows: The CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory. Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU). This scheme is shown in Figure 9.21.

9.6.1.1 IA-32 Segmentation

The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K. The logical address space of a process is divided into two partitions. The first partition consists of up to 8 K segments that are private to that process. The second partition consists of up to 8 K segments that are shared among all the processes. Information about the first partition is kept in the **local descriptor table (LDT)**; information about the second partition is kept in the **global descriptor table (GDT)**. Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.

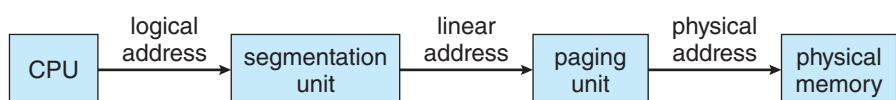
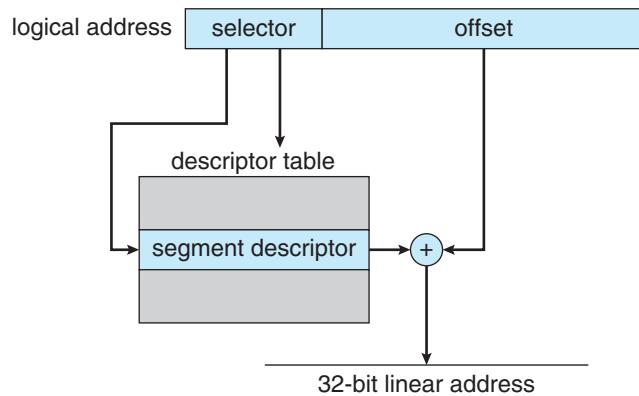


Figure 9.21 Logical to physical address translation in IA-32.

**Figure 9.22** IA-32 segmentation.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

Here, *s* designates the segment number, *g* indicates whether the segment is in the GDT or LDT, and *p* deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It also has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or the GDT. This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

The linear address on the IA-32 is 32 bits long and is formed as follows. The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question is used to generate a **linear address**. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This is shown in Figure 9.22. In the following section, we discuss how the paging unit turns this linear address into a physical address.

9.6.1.2 IA-32 Paging

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:

page number		page offset
<i>p</i> ₁	<i>p</i> ₂	<i>d</i>
10	10	12

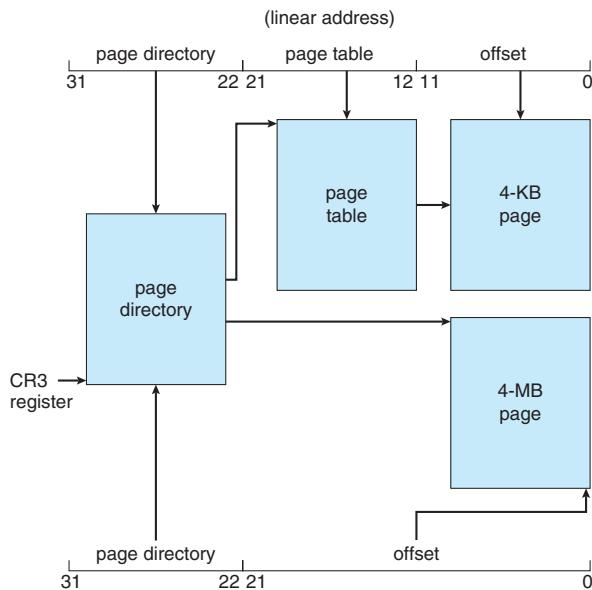


Figure 9.23 Paging in the IA-32 architecture.

The address-translation scheme for this architecture is similar to the scheme shown in Figure 9.16. The IA-32 address translation is shown in more detail in Figure 9.23. The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **page directory**. (The CR3 register points to the page directory for the current process.) The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address. Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.

One entry in the page directory is the **Page_Size** flag, which—if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB. If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table. The table can then be brought into memory on demand.

As software developers began to discover the 4-GB memory limitations of 32-bit architectures, Intel adopted a **page address extension (PAE)**, which allows 32-bit processors to access a physical address space larger than 4 GB. The fundamental difference introduced by PAE support was that paging went from a two-level scheme (as shown in Figure 9.23) to a three-level scheme, where the top two bits refer to a **page directory pointer table**. Figure 9.24 illustrates a PAE system with 4-KB pages. (PAE also supports 2-MB pages.)

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to

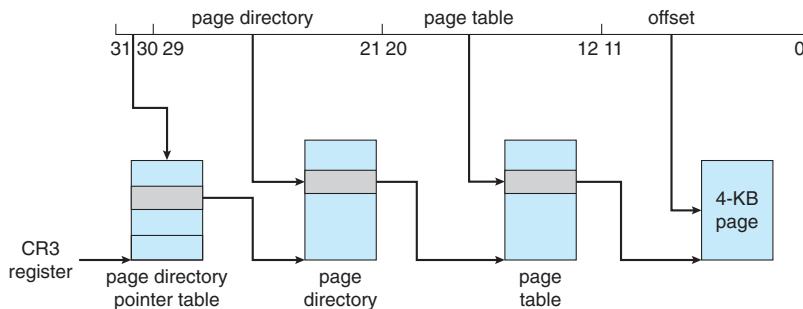


Figure 9.24 Page address extensions.

extend from 20 to 24 bits. Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory. It is important to note that operating system support is required to use PAE. Both Linux and macOS support PAE. However, 32-bit versions of Windows desktop operating systems still provide support for only 4 GB of physical memory, even if PAE is enabled.

9.6.2 x86-64

Intel has had an interesting history of developing 64-bit architectures. Its initial entry was the IA-64 (later named [Itanium](#)) architecture, but that architecture was not widely adopted. Meanwhile, another chip manufacturer—AMD—began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set. The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances. Historically, AMD had often developed chips based on Intel's architecture, but now the roles were reversed as Intel adopted AMD's x86-64 architecture. In discussing this architecture, rather than using the commercial names [AMD64](#) and [Intel 64](#), we will use the more general term [x86-64](#).

Support for a 64-bit address space yields an astonishing 2^{64} bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes). However, even though 64-bit systems can potentially address this much memory, in practice far fewer than 64 bits are used for address representation in current designs. The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy. The representation of the linear address appears in Figure 9.25. Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4,096 terabytes).

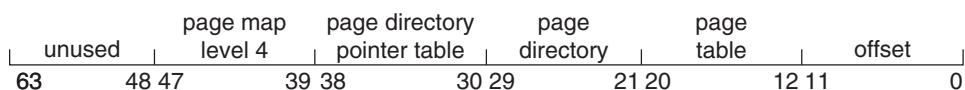


Figure 9.25 x86-64 linear address.

9.7 Example: ARMv8 Architecture

Although Intel chips have dominated the personal computer market for more than 30 years, chips for mobile devices such as smartphones and tablet computers often instead run on ARM processors. Interestingly, whereas Intel both designs and manufactures chips, ARM only designs them. It then licenses its architectural designs to chip manufacturers. Apple has licensed the ARM design for its iPhone and iPad mobile devices, and most Android-based smartphones use ARM processors as well. In addition to mobile devices, ARM also provides architecture designs for real-time embedded systems. Because of the abundance of devices that run on the ARM architecture, over 100 billion ARM processors have been produced, making it the most widely used architecture when measured in the quantity of chips produced. In this section, we describe the 64-bit ARMv8 architecture.

The ARMv8 has three different **translation granules**: 4 KB, 16 KB, and 64 KB. Each translation granule provides different page sizes, as well as larger sections of contiguous memory, known as **regions**. The page and region sizes for the different translation granules are shown below:

Translation Granule Size	Page Size	Region Size
4 KB	4 KB	2 MB, 1 GB
16 KB	16 KB	32 MB
64 KB	64 KB	512 MB

For 4-KB and 16-KB granules, up to four levels of paging may be used, with up to three levels of paging for 64-KB granules. Figure 9.26 illustrates the ARMv8 address structure for the 4-KB translation granule with up to four levels of paging. (Notice that although ARMv8 is a 64-bit architecture, only 48 bits are currently used.) The four-level hierarchical paging structure for the 4-KB translation granule is illustrated in Figure 9.27. (The TTBR register is the **translation table base register** and points to the level 0 table for the current thread.)

If all four levels are used, the offset (bits 0–11 in Figure 9.26) refers to the offset within a 4-KB page. However, notice that the table entries for level 1 and

64-BIT COMPUTING

History has taught us that even though memory capacities, CPU speeds, and similar computer capabilities seem large enough to satisfy demand for the foreseeable future, the growth of technology ultimately absorbs available capacities, and we find ourselves in need of additional memory or processing power, often sooner than we think. What might the future of technology bring that would make a 64-bit address space seem too small?

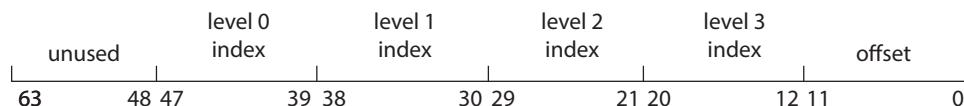


Figure 9.26 ARM 4-KB translation granule.

level 2 may refer either to another table or to a 1-GB region (level-1 table) or 2-MB region (level-2 table). As an example, if the level-1 table refers to a 1-GB region rather than a level-2 table, the low-order 30 bits (bits 0–29 in Figure 9.26) are used as an offset into this 1-GB region. Similarly, if the level-2 table refers to a 2-MB region rather than a level-3 table, the low-order 21 bits (bits 0–20 in Figure 9.26) refer to the offset within this 2-MB region.

The ARM architecture also supports two levels of TLBs. At the inner level are two **micro TLBs**—a TLB for data and another for instructions. The micro TLB supports ASIDs as well. At the outer level is a single **main TLB**. Address translation begins at the micro-TLB level. In the case of a miss, the main TLB is then checked. If both TLBs yield misses, a page table walk must be performed in hardware.

9.8 Summary

- Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address.
- One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.

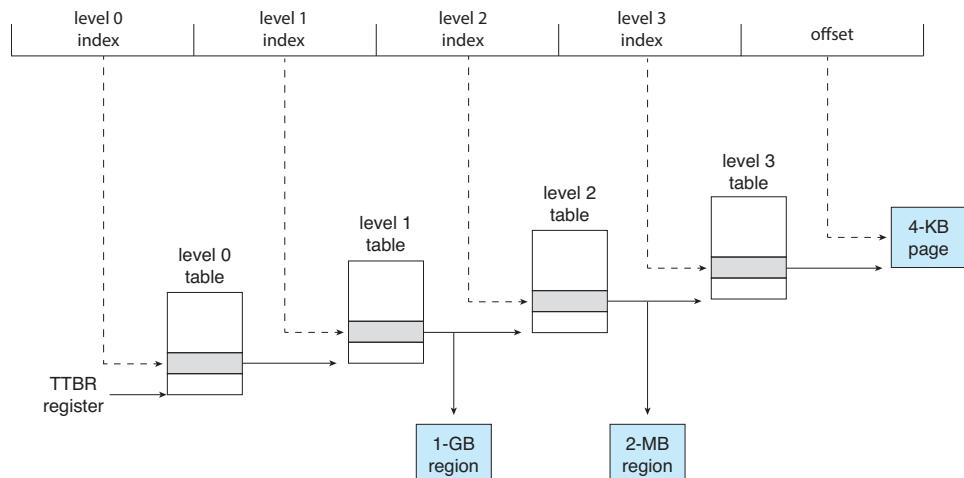


Figure 9.27 ARM four-level hierarchical paging.

- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.
- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per-process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.
- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.
- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.
- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.
- The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports page-address extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.

Practice Exercises

- 9.1 Name two differences between logical and physical addresses.
- 9.2 Why are page sizes always powers of 2?
- 9.3 Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base-limit register pairs are provided: one for instructions and one for data. The instruction

base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

- 9.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
 - a. How many bits are there in the logical address?
 - b. How many bits are there in the physical address?
 - 9.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?
 - 9.6 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?
 - 9.7 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
 - a. 3085
 - b. 42095
 - c. 215201
 - d. 650000
 - e. 2000001
 - 9.8 The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?
 - a. A conventional, single-level page table
 - b. An inverted page table
- What is the maximum amount of physical memory in the BTV operating system?
- 9.9 Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
 - a. How many bits are required in the logical address?
 - b. How many bits are required in the physical address?
 - 9.10 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?
 - a. A conventional, single-level page table
 - b. An inverted page table

Further Reading

The concept of paging can be credited to the designers of the Atlas system, which has been described by [Kilburn et al. (1961)] and by [Howarth et al. (1961)].

[Hennessy and Patterson (2012)] explain the hardware aspects of TLBs, caches, and MMUs. [Jacob and Mudge (2001)] describe techniques for managing the TLB. [Fang et al. (2001)] evaluate support for large pages.

PAE support for Windows systems is discussed in <http://msdn.microsoft.com/en-us/library/windows/hardware/gg487512.aspx>. An overview of the ARM architecture is provided in <http://www.arm.com/products/processors/cortex-a/cortex-a9.php>

Bibliography

[Fang et al. (2001)] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, “Reevaluating Online Superpage Promotion with Hardware Support”, *Proceedings of the International Symposium on High-Performance Computer Architecture*, Volume 50, Number 5 (2001).

[Hennessy and Patterson (2012)] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth Edition, Morgan Kaufmann (2012).

[Howarth et al. (1961)] D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.

[Jacob and Mudge (2001)] B. Jacob and T. Mudge, “Uniprocessor Virtual Memory Without TLBs”, *IEEE Transactions on Computers*, Volume 50, Number 5 (2001).

[Kilburn et al. (1961)] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.

Chapter 9 Exercises

- 9.11 Explain the difference between internal and external fragmentation.
- 9.12 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?
- 9.13 Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.
- 9.14 Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?
 - a. Contiguous memory allocation
 - b. Paging
- 9.15 Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:
 - a. External fragmentation
 - b. Internal fragmentation
 - c. Ability to share code across processes
- 9.16 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?
- 9.17 Explain why mobile operating systems such as iOS and Android do not support swapping.
- 9.18 Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?
- 9.19 Explain why address-space identifiers (ASIDs) are used in TLBs.
- 9.20 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

- a. Contiguous memory allocation
 - b. Paging
- 9.21** Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?
- a. 21205
 - b. 164250
 - c. 121357
 - d. 16479315
 - e. 27253187
- 9.22** The MPV operating system is designed for embedded systems and has a 24-bit virtual address, a 20-bit physical address, and a 4-KB page size. How many entries are there in each of the following?
- a. A conventional, single-level page table
 - b. An inverted page table
- What is the maximum amount of physical memory in the MPV operating system?
- 9.23** Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.
- a. How many bits are required in the logical address?
 - b. How many bits are required in the physical address?
- 9.24** Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?
- a. A conventional, single-level page table
 - b. An inverted page table
- 9.25** Consider a paging system with the page table stored in memory.
- a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?
 - b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)
- 9.26** What is the purpose of paging the page tables?
- 9.27** Consider the IA-32 address-translation scheme shown in Figure 9.22.
- a. Describe all the steps taken by the IA-32 in translating a logical address into a physical address.
 - b. What are the advantages to the operating system or hardware that provides such complicated memory translation?

- c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?

Programming Problems

- 9.28 Assume that a system has a 32-bit virtual address with a 4-KB page size. Write a C program that is passed a virtual address (in decimal) on the command line and have it output the page number and offset for the given address. As an example, your program would run as follows:

```
./addresses 19986
```

Your program would output:

```
The address 19986 contains:  
page number = 4  
offset = 3602
```

Writing this program will require using the appropriate data type to store 32 bits. We encourage you to use unsigned data types as well.

Programming Projects

Contiguous Memory Allocation

In Section 9.2, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size *MAX* where addresses may range from 0 ... *MAX* – 1. Your program must respond to four different requests:

1. Request for a contiguous block of memory
2. Release of a contiguous block of memory
3. Compact unused holes of memory into one single block
4. Report the regions of free and allocated memory

Your program will be passed the initial amount of memory at startup. For example, the following initializes the program with 1 MB (1,048,576 bytes) of memory:

```
./allocator 1048576
```

Once your program has started, it will present the user with the following prompt:

```
allocator>
```

It will then respond to the following commands: RQ (request), RL (release), C (compact), STAT (status report), and X (exit).

A request for 40,000 bytes will appear as follows:

```
allocator>RQ P0 40000 W
```

The first parameter to the RQ command is the new process that requires the memory, followed by the amount of memory being requested, and finally the strategy. (In this situation, “W” refers to worst fit.)

Similarly, a release will appear as:

```
allocator>RL P0
```

This command will release the memory that has been allocated to process P0.

The command for compaction is entered as:

```
allocator>C
```

This command will compact unused holes of memory into one region.

Finally, the STAT command for reporting the status of memory is entered as:

```
allocator>STAT
```

Given this command, your program will report the regions of memory that are allocated and the regions that are unused. For example, one possible arrangement of memory allocation would be as follows:

```
Addresses [0:315000] Process P1
Addresses [315001: 512500] Process P3
Addresses [512501:625575] Unused
Addresses [625575:725100] Process P6
Addresses [725001] . . .
```

Allocating Memory

Your program will allocate memory using one of the three approaches highlighted in Section 9.2.2, depending on the flag that is passed to the RQ command. The flags are:

- F—first fit
- B—best fit
- W—worst fit

This will require that your program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy. If there is insufficient memory to allocate to a request, it will output an error message and reject the request.

Your program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the STAT command and is also needed when memory is released via the RL command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

Compaction

If the user enters the C command, your program will compact the set of holes into one larger hole. For example, if you have four separate holes of size 550 KB, 375 KB, 1,900 KB, and 4,500 KB, your program will combine these four holes into one large hole of size 7,325 KB.

There are several strategies for implementing compaction, one of which is suggested in Section 9.2.3. Be sure to update the beginning address of any processes that have been affected by compaction.

Virtual Memory



In Chapter 9, we discussed various memory-management strategies used in computer systems. All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. However, they tend to require that an entire process be in memory before it can execute.

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the programmer from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files and libraries, and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we provide a detailed overview of virtual memory, examine how it is implemented, and explore its complexity and benefits.

CHAPTER OBJECTIVES

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

10.1 Background

The memory-management algorithms outlined in Chapter 9 are necessary because of one basic requirement: the instructions being executed must be in

physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.

The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

Even in those cases where the entire program is needed, it may not all be needed at the same time.

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large *virtual* address space, simplifying the programming task.
- Because each program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and its users.

Virtual memory involves the separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 10.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on programming the problem that is to be solved.

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure 10.2. Recall from Chapter 9, though, that in fact physical memory is organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory-

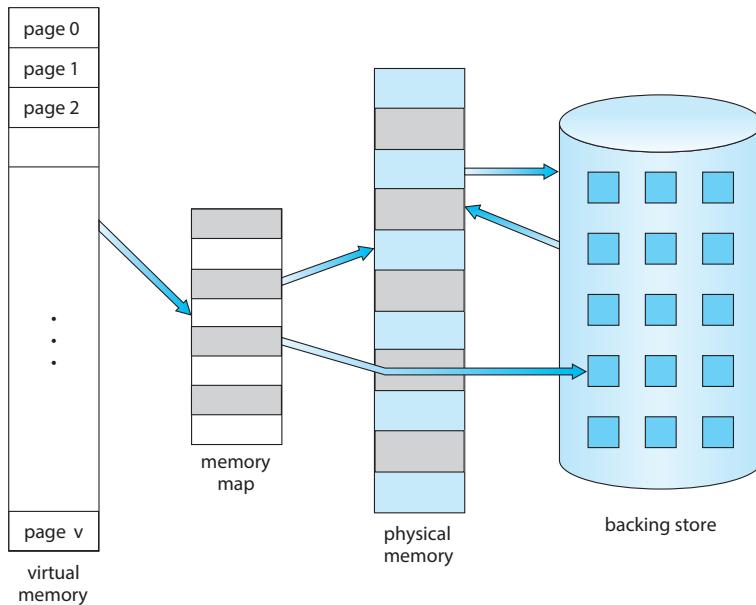


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

management unit (MMU) to map logical pages to physical page frames in memory.

Note in Figure 10.2 that we allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse** address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

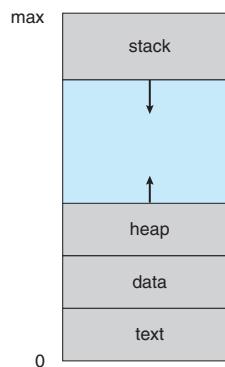


Figure 10.2 Virtual address space of a process in memory.

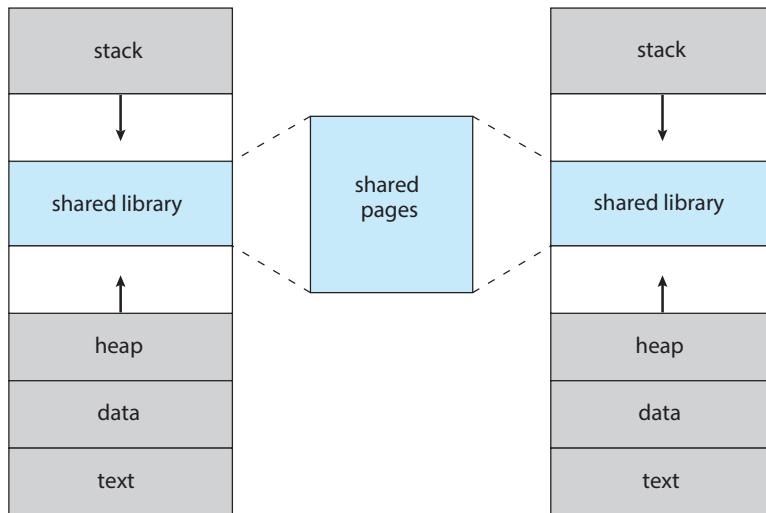


Figure 10.3 Shared library using virtual memory.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing (Section 9.3.4). This leads to the following benefits:

- System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure 10.3). Typically, a library is mapped read-only into the space of each process that is linked with it.
- Similarly, processes can share memory. Recall from Chapter 3 that two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure 10.3.
- Pages can be shared during process creation with the `fork()` system call, thus speeding up process creation.

We further explore these—and other—benefits of virtual memory later in this chapter. First, though, we discuss implementing virtual memory through demand paging.

10.2 Demand Paging

Consider how an executable program might be loaded from secondary storage into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that

we may not initially *need* the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for *all* options, regardless of whether or not an option is ultimately selected by the user.

An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are *demanded* during program execution. Pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping (Section 9.5.2) where processes reside in secondary memory (usually an HDD or NVM device). Demand paging explains one of the primary benefits of virtual memory—by loading only the portions of programs that are *needed*, memory is used more efficiently.

10.2.1 Basic Concepts

The general concept behind demand paging, as mentioned, is to load a page in memory only when it is needed. As a result, while a process is executing, some pages will be in memory, and some will be in secondary storage. Thus, we need some form of hardware support to distinguish between the two. The valid-invalid bit scheme described in Section 9.3.3 can be used for this purpose. This

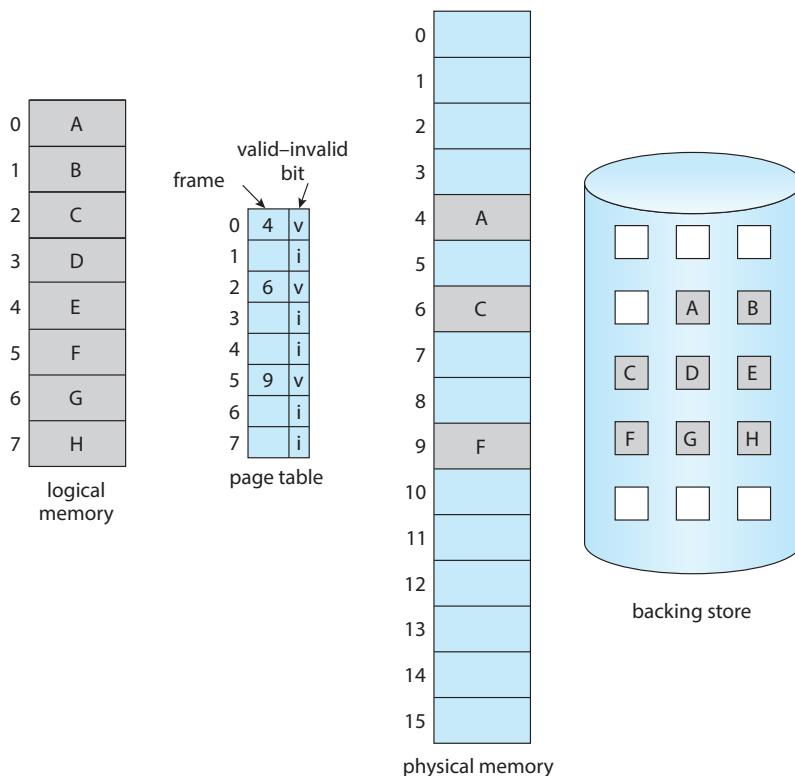


Figure 10.4 Page table when some pages are not in main memory.

time, however, when the bit is set to “valid,” the associated page is both legal and in memory. If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently in secondary storage. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid. This situation is depicted in Figure 10.4. (Notice that marking a page invalid will have no effect if the process never attempts to access that page.)

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system’s failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 10.5):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).

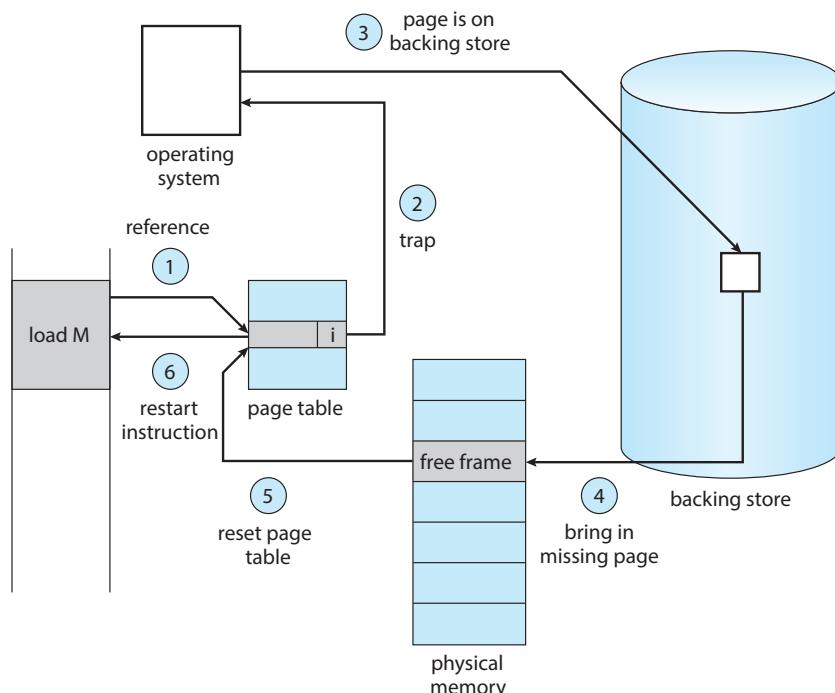


Figure 10.5 Steps in handling a page fault.

4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with *no* pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference**, described in Section 10.6.1, which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table.** This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
- **Secondary memory.** This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk or NVM device. It is known as the swap device, and the section of storage used for this purpose is known as **swap space**. Swap-space allocation is discussed in Chapter 11.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in *exactly* the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.

3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty arises when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place; we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

This is by no means the only architectural problem resulting from adding paging to an existing architecture to allow demand paging, but it illustrates some of the difficulties involved. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to a process. Thus, people often assume that paging can be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

10.2.2 Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory. To resolve page faults, most operating systems maintain a **free-frame list**, a pool of free frames for satisfying such requests (Figure 10.6). (Free frames must also be allocated when the stack or heap segments from a process expand.) Operating systems typically allo-



Figure 10.6 List of free frames.

cate free frames using a technique known as **zero-fill-on-demand**. Zero-fill-on-demand frames are “zeroed-out” before being allocated, thus erasing their previous contents. (Consider the potential security implications of *not* clearing out the contents of a frame before reassigning it.)

When a system starts up, all available memory is placed on the free-frame list. As free frames are requested (for example, through demand paging), the size of the free-frame list shrinks. At some point, the list either falls to zero or falls below a certain threshold, at which point it must be repopulated. We cover strategies for both of these situations in Section 10.4.

10.2.3 Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let’s compute the **effective access time** for a demand-paged memory. Assume the memory-access time, denoted ma , is 10 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.

Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The effective access time is then

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal, and determine the location of the page in secondary storage.
5. Issue a read from the storage to a free frame:
 - a. Wait in a queue until the read request is serviced.
 - b. Wait for the device seek and/or latency time.
 - c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU core to some other process.
7. Receive an interrupt from the storage I/O subsystem (I/O completed).
8. Save the registers and process state for the other process (if step 6 is executed).
9. Determine that the interrupt was from the secondary storage device.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU core to be allocated to this process again.

12. Restore the registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, there are three major task components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. Let's consider the case of HDDs being used as the paging device. The page-switch time will probably be close to 8 milliseconds. (A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is about 8 milliseconds, including hardware and software time.) Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device, we have to add queuing time as we wait for the paging device to be free to service our request, increasing even more the time to page in.

With an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (200) + p \text{ (8 milliseconds)} \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + 7,999,800 \times p.\end{aligned}$$

We see, then, that the effective access time is directly proportional to the **page-fault rate**. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging! If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

$$\begin{aligned}220 &> 200 + 7,999,800 \times p, \\ 20 &> 7,999,800 \times p, \\ p &< 0.0000025.\end{aligned}$$

That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer than one memory access out of 399,990 to page-fault. In sum, it is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

An additional aspect of demand paging is the handling and overall use of swap space. I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used (Chapter 11). One option for the

system to gain better paging throughput is by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space. The obvious disadvantage of this approach is the copying of the file image at program start-up. A second option—and one practiced by several operating systems, including Linux and Windows—is to demand-page from the file system initially but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are read from the file system but that all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space used through demand paging of binary executable files. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these frames can simply be overwritten (because they are never modified), and the pages can be read in from the file system again if needed. Using this approach, the file system itself serves as the backing store. However, swap space must still be used for pages not associated with a file (known as **anonymous memory**); these pages include the stack and heap for a process. This method appears to be a good compromise and is used in several systems, including Linux and BSD UNIX.

As described in Section 9.5.3, mobile operating systems typically do not support swapping. Instead, these systems demand-page from the file system and reclaim read-only pages (such as code) from applications if memory becomes constrained. Such data can be demand-paged from the file system if it is later needed. Under iOS, anonymous memory pages are never reclaimed from an application unless the application is terminated or explicitly releases the memory. In Section 10.7, we cover compressed memory, a commonly used alternative to swapping in mobile systems.

10.3 Copy-on-Write

In Section 10.2, we illustrated how a process can start quickly by demand-paging in the page containing the first instruction. However, process creation using the `fork()` system call may initially bypass the need for demand paging by using a technique similar to page sharing (covered in Section 9.3.4). This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

Recall that the `fork()` system call creates a child process that is a duplicate of its parent. Traditionally, `fork()` worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the `exec()` system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as **copy-on-write**, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures 10.7 and 10.8, which show the contents of the physical memory before and after process 1 modifies page C.

For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will obtain a frame from the free-frame list and create a copy

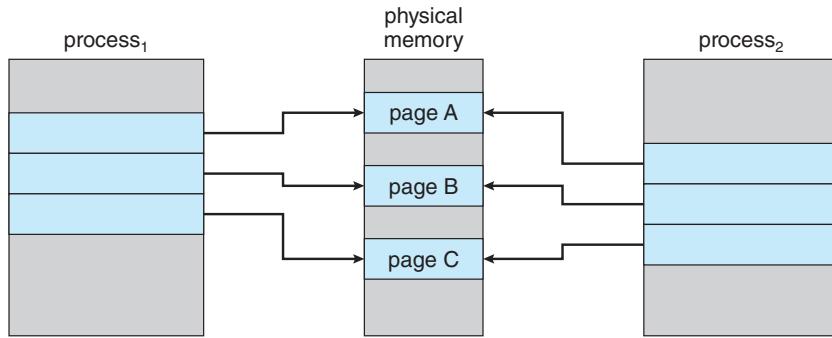


Figure 10.7 Before process 1 modifies page C.

of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes. Note, too, that only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child. Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS.

Several versions of UNIX (including Linux, macOS, and BSD UNIX) provide a variation of the `fork()` system call—`vfork()` (for **virtual memory fork**)—that operates differently from `fork()` with copy-on-write. With `vfork()`, the parent process is suspended, and the child process uses the address space of the parent. Because `vfork()` does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, `vfork()` must be used with caution to ensure that the child process does not modify the address space of the parent. `vfork()` is intended to be used when the child process calls `exec()` immediately after creation. Because no copying of pages takes place, `vfork()`

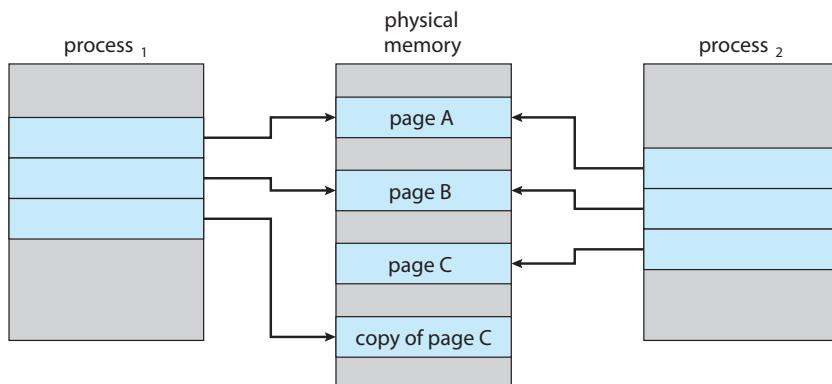


Figure 10.8 After process 1 modifies page C.

is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

10.4 Page Replacement

In our earlier discussion of the page-fault rate, we assumed that each page faults at most once, when it is first referenced. This representation is not strictly accurate, however. If a process of ten pages actually uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).

If we increase our degree of multiprogramming, we are **over-allocating** memory. If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a considerable amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both processes and the I/O subsystem to compete for all system memory. Section 14.6 discusses the integrated relationship between I/O buffers and virtual memory techniques.

Over-allocation of memory manifests itself as follows. While a process is executing, a page fault occurs. The operating system determines where the desired page is residing on secondary storage but then finds that there are *no* free frames on the free-frame list; all memory is in use. This situation is illustrated in Figure 10.9, where the fact that there are no free frames is depicted by a question mark.

The operating system has several options at this point. It could terminate the process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system—paging should be logically transparent to the user. So this option is not the best choice.

The operating system could instead use standard swapping and swap out a process, freeing all its frames and reducing the level of multiprogramming. However, as discussed in Section 9.5, standard swapping is no longer used by most operating systems due to the overhead of copying entire processes between memory and swap space. Most operating systems now combine swapping pages with **page replacement**, a technique we describe in detail in the remainder of this section.

10.4.1 Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its

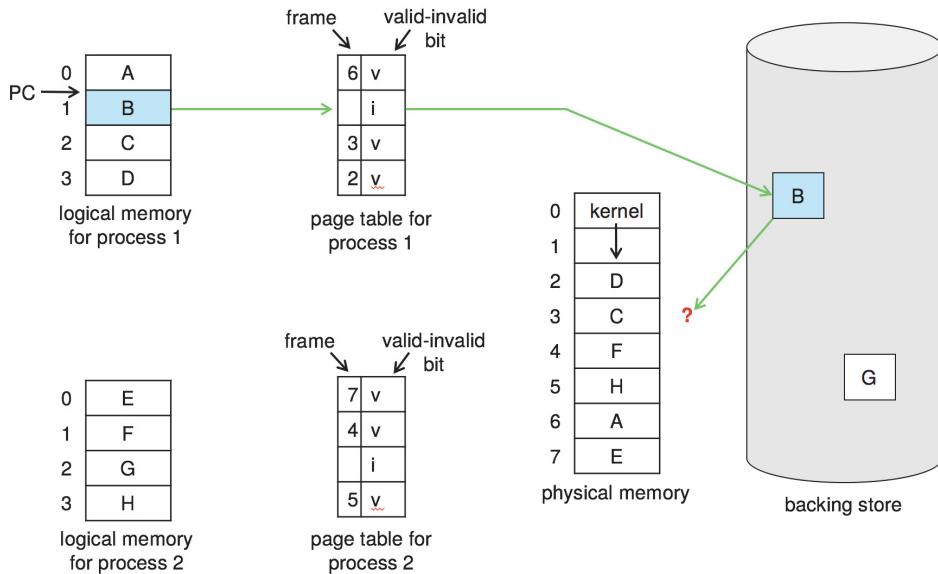


Figure 10.9 Need for page replacement.

contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 10.10). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on secondary storage.
2. Find a free frame:
 - a. If there is a free frame, use it.
 - b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.
 - c. Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

Notice that, if no frames are free, *two* page transfers (one for the page-out and one for the page-in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

We can reduce this overhead by using a **modify bit** (or **dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set,

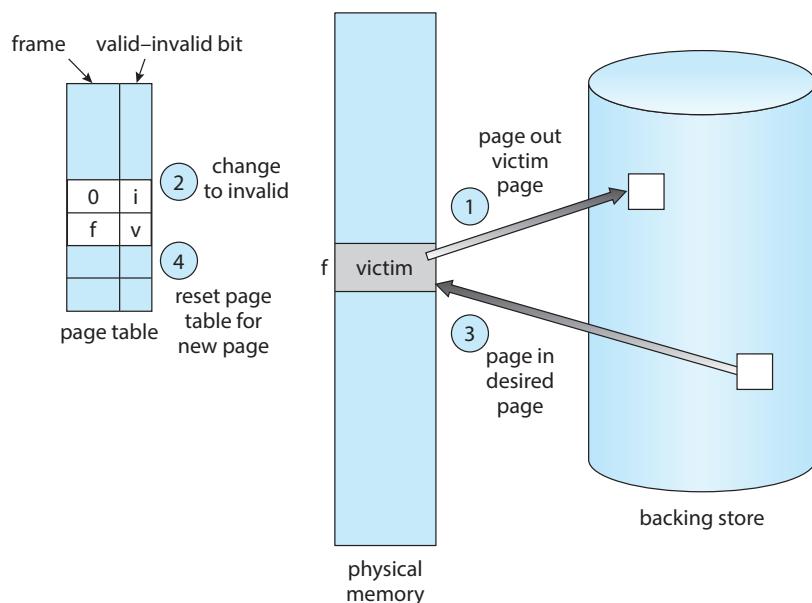


Figure 10.10 Page replacement.

we know that the page has been modified since it was read in from secondary storage. In this case, we must write the page to storage. If the modify bit is not set, however, the page has *not* been modified since it was read into memory. In this case, we need not write the memory page to storage: it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half *if* the page has not been modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With no demand paging, logical addresses are mapped into physical addresses, and the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a process of twenty pages, we can execute it in ten frames simply by using demand paging and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to secondary storage. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because secondary storage

I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**. We can generate reference strings artificially (by using a random-number generator, for example), or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we use two facts.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page p , then any references to page p that *immediately* follow will never cause a page fault. Page p will be in memory after the first reference, so the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

```
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,  
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
```

At 100 bytes per page, this sequence is reduced to the following reference string:

```
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
```

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults—one fault for the first reference to each page. In contrast, with only one frame available, we would have a replacement with every reference, resulting in eleven faults. In general, we expect a curve such as that in Figure 10.11. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

We next illustrate several page-replacement algorithms. In doing so, we use the reference string

```
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
```

for a memory with three frames.

10.4.2 FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a

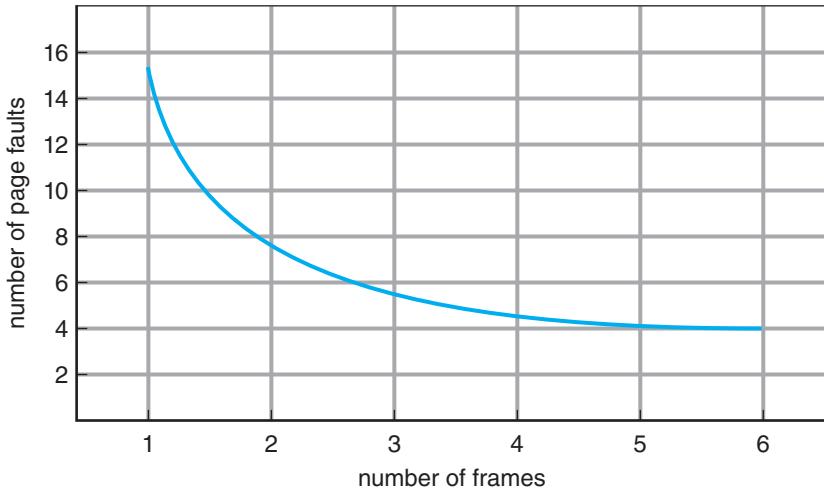


Figure 10.11 Graph of page faults versus number of frames.

page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 10.12. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new

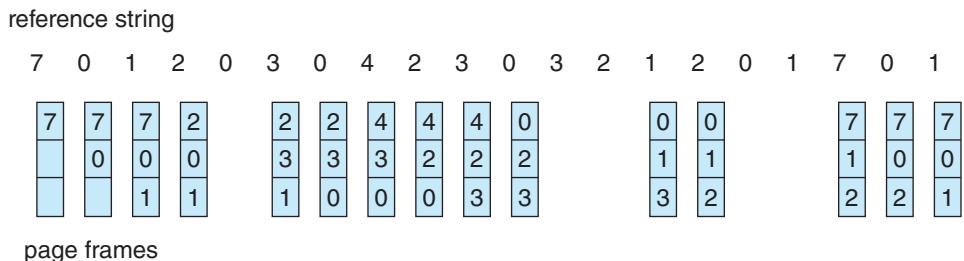


Figure 10.12 FIFO page-replacement algorithm.

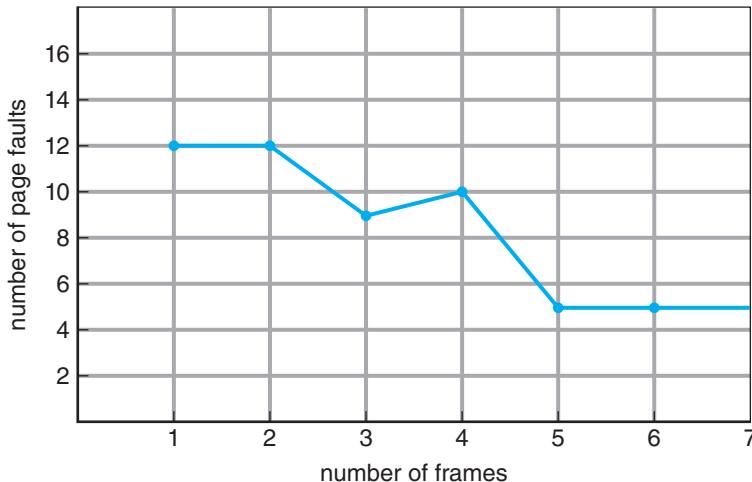


Figure 10.13 Page-fault curve for FIFO replacement on a reference string.

one, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Figure 10.13 shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly**: for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

10.4.3 Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an **optimal page-replacement algorithm**—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 10.14. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas

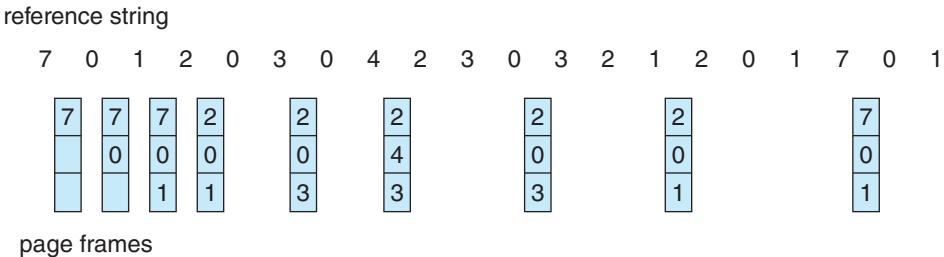


Figure 10.14 Optimal page-replacement algorithm.

page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm in Section 5.3.2.) As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

10.4.4 LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be *used*. If we use the recent past as an approximation of the near future, then we can replace the page that *has not been used* for the longest period of time. This approach is the **least recently used (LRU) algorithm**.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let S^R be the reverse of a reference string S , then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on S^R . Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on S^R .)

The result of applying LRU replacement to our example reference string is shown in Figure 10.15. The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults

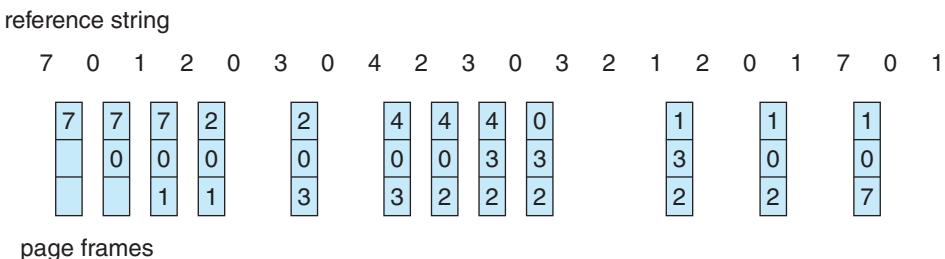


Figure 10.15 LRU page-replacement algorithm.

for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is *how* to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- **Counters.** In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the “time” of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.
- **Stack.** Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom (Figure 10.16). Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Like optimal replacement, LRU replacement does not suffer from Belady’s anomaly. Both belong to a class of page-replacement algorithms, called **stack algorithms**, that can never exhibit Belady’s anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a *subset* of the set of pages that would be in memory with n

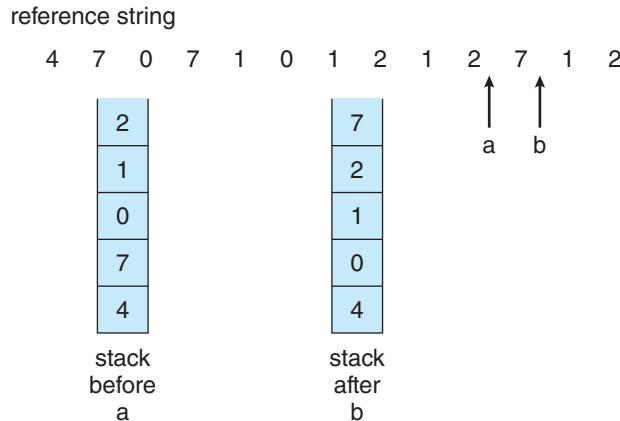


Figure 10.16 Use of a stack to record the most recent page references.

+ 1 frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for *every* memory reference. If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every process by a factor of ten. Few systems could tolerate that level of overhead for memory management.

10.4.5 LRU-Approximation Page Replacement

Not many computer systems provide sufficient hardware support for true LRU page replacement. In fact, some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a **reference bit**. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the *order* of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

10.4.5.1 Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right

by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods. A page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them.

The number of bits of history included in the shift register can be varied, of course, and is selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance page-replacement algorithm**.

10.4.5.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance algorithm (sometimes referred to as the **clock** algorithm) is as a circular queue. A pointer (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 10.17). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

10.4.5.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit (described in Section 10.4.1) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon

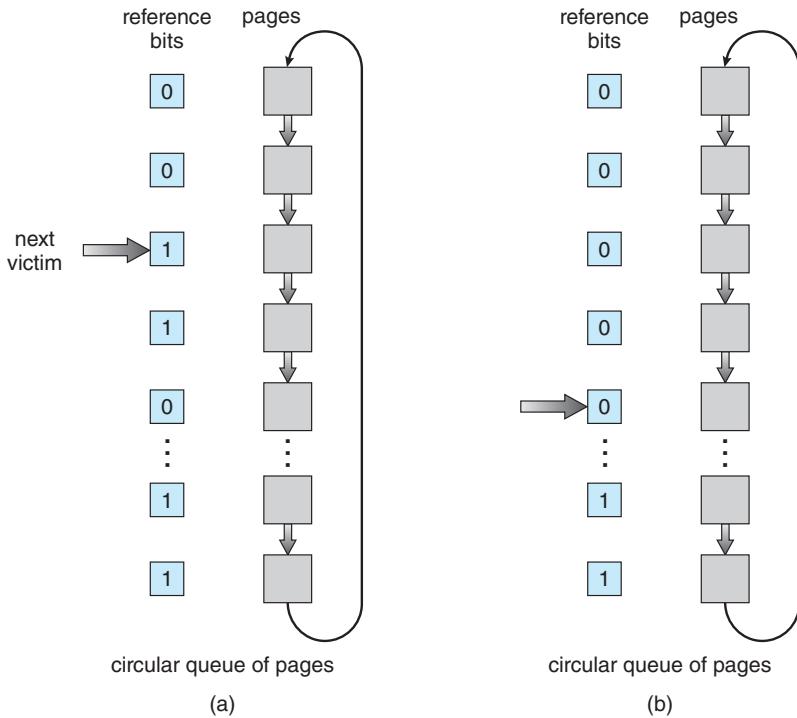


Figure 10.17 Second-chance (clock) page-replacement algorithm.

4. (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to secondary storage before it can be replaced

Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified in order to reduce the number of I/Os required.

10.4.6 Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of

a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

- The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

10.4.7 Page-Buffering Algorithms

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to secondary storage. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to secondary storage, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

Some versions of the UNIX system use this method in conjunction with the second-chance algorithm. It can be a useful augmentation to any page-replacement algorithm, to reduce the penalty incurred if the wrong victim page is selected. We describe these—and other—modifications in Section 10.5.3.

10.4.8 Applications and Page Replacement

In certain cases, applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all. A typical example is a database, which provides its own memory management and I/O buffering. Applications like this understand their memory use and storage use better than does an operating system that is implementing algorithms for general-purpose use. Furthermore, if the operating system is buffering I/O and the application is doing so as well, then twice the memory is being used for a set of I/O.

In another example, data warehouses frequently perform massive sequential storage reads, followed by computations and writes. The LRU algorithm

would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones (as it starts its sequential reads again). Here, MFU would actually be more efficient than LRU.

Because of such problems, some operating systems give special programs the ability to use a secondary storage partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed raw I/O. Raw I/O bypasses all the file-system services, such as file I/O demand paging, file locking, prefetching, space allocation, file names, and directories. Note that although certain applications are more efficient when implementing their own special-purpose storage services on a raw partition, most applications perform better when they use the regular file-system services.

10.5 Allocation of Frames

We turn next to the issue of allocation. How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

Consider a simple case of a system with 128 frames. The operating system may take 35, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy. We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the storage device as the user process continues to execute. Other variants are also possible, but the basic strategy is clear: the user process is allocated any free frame.

10.5.1 Minimum Number of Frames

Our strategies for the allocation of frames are constrained in various ways. We cannot, for example, allocate more than the total number of available frames (unless there is page sharing). We must also allocate at least a minimum number of frames. Here, we look more closely at the latter requirement.

One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. In addition, remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

For example, consider a machine in which all memory-reference instructions may reference only one memory address. In this case, we need at least one frame for the instruction and one frame for the memory reference. In addition, if one-level indirect addressing is allowed (for example, a load instruction on frame 16 can refer to an address on frame 0, which is an indirect reference to frame 23), then paging requires at least three frames per process. (Think about what might happen if a process had only two frames.)

The minimum number of frames is defined by the computer architecture. For example, if the move instruction for a given architecture includes more than one word for some addressing modes, the instruction itself may straddle two frames. In addition, if each of its two operands may be indirect references, a total of six frames are required. As another example, the move instruction for Intel 32- and 64-bit architectures allows data to move only from register to register and between registers and memory; it does not allow direct memory-to-memory movement, thereby limiting the required minimum number of frames for a process.

Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory. In between, we are still left with significant choice in frame allocation.

10.5.2 Allocation Algorithms

The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames (ignoring frames needed by the operating system for the moment). For instance, if there are 93 frames and 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffer pool. This scheme is called **equal allocation**.

An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

To solve this problem, we can use **proportional allocation**, in which we allocate available memory to each process according to its size. Let the size of the virtual memory for process p_i be s_i , and define

$$S = \sum s_i.$$

Then, if the total number of available frames is m , we allocate a_i frames to process p_i , where a_i is approximately

$$a_i = s_i / S \times m.$$

Of course, we must adjust each a_i to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m .

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

$$10/137 \times 62 \approx 4 \text{ and}$$

$$127/137 \times 62 \approx 57.$$

In this way, both processes share the available frames according to their “needs,” rather than equally.

In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level. If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process. Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

10.5.3 Global versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme wherein we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process. Whereas with a local replacement strategy, the number of frames allocated to a process does not change, with global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose *its* frames for replacement).

One problem with a global replacement algorithm is that the set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (for example, taking 0.5 seconds for one execution and 4.3 seconds for the next execution) because of totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory. Thus, global replacement generally results in greater system throughput. It is therefore the more commonly used method.

MAJOR AND MINOR PAGE FAULTS

As described in Section 10.2.1, a page fault occurs when a page does not have a valid mapping in the address space of a process. Operating systems generally distinguish between two types of page faults: **major** and **minor** faults. (Windows refers to major and minor faults as **hard** and **soft** faults, respectively.) A major page fault occurs when a page is referenced and the page is not in memory. Servicing a major page fault requires reading the desired page from the backing store into a free frame and updating the page table. Demand paging typically generates an initially high rate of major page faults.

Minor page faults occur when a process does not have a logical mapping to a page, yet that page is in memory. Minor faults can occur for one of two reasons. First, a process may reference a shared library that is in memory, but the process does not have a mapping to it in its page table. In this instance, it is only necessary to update the page table to refer to the existing page in memory. A second cause of minor faults occurs when a page is reclaimed from a process and placed on the free-frame list, but the page has not yet been zeroed out and allocated to another process. When this kind of fault occurs, the frame is removed from the free-frame list and reassigned to the process. As might be expected, resolving a minor page fault is typically much less time consuming than resolving a major page fault.

You can observe the number of major and minor page faults in a Linux system using the command `ps -eo min_flt,maj_flt,cmd`, which outputs the number of minor and major page faults, as well as the command that launched the process. An example output of this `ps` command appears below:

MINFL	MAJFL	CMD
186509	32	/usr/lib/systemd/systemd-logind
76822	9	/usr/sbin/sshd -D
1937	0	vim 10.tex
699	14	/sbin/auditd -n

Here, it is interesting to note that, for most commands, the number of major page faults is generally quite low, whereas the number of minor faults is much higher. This indicates that Linux processes likely take significant advantage of shared libraries as, once a library is loaded in memory, subsequent page faults are only minor faults.

Next, we focus on one possible strategy that we can use to implement a global page-replacement policy. With this approach, we satisfy all memory requests from the free-frame list, but rather than waiting for the list to drop to zero before we begin selecting pages for replacement, we trigger page replacement when the list falls below a certain threshold. This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Such a strategy is depicted in Figure 10.18. The strategy's purpose is to keep the amount of free memory above a minimum threshold. When it drops

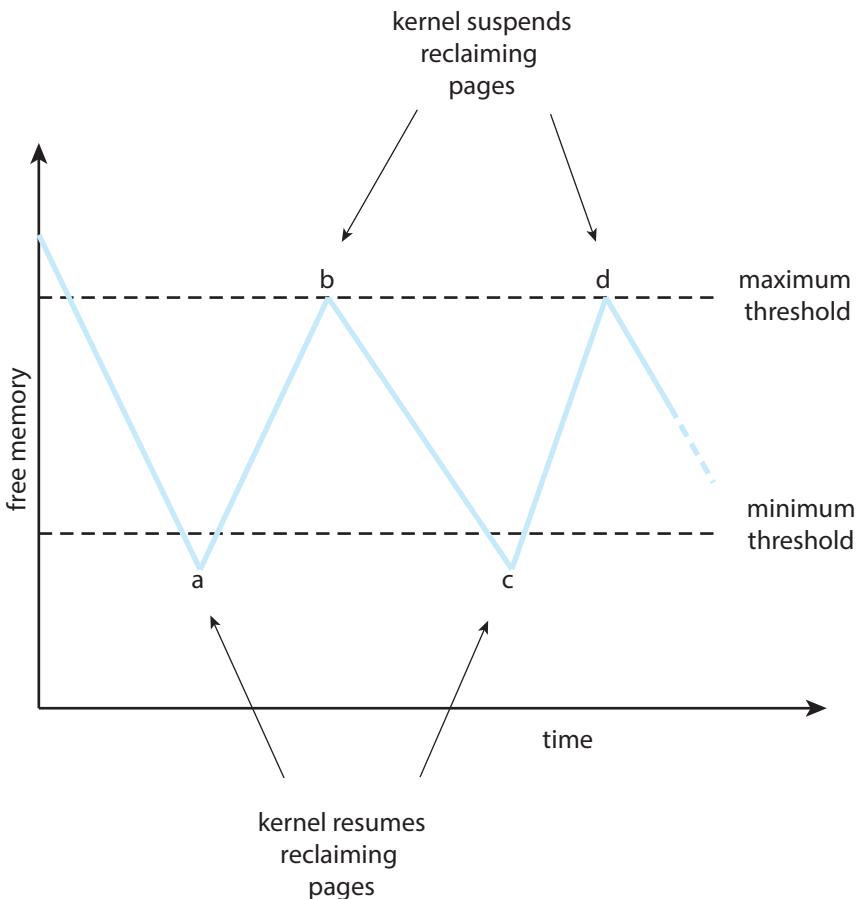


Figure 10.18 Reclaiming pages.

below this threshold, a kernel routine is triggered that begins reclaiming pages from all processes in the system (typically excluding the kernel). Such kernel routines are often known as **reapers**, and they may apply any of the page-replacement algorithms covered in Section 10.4. When the amount of free memory reaches the maximum threshold, the reaper routine is suspended, only to resume once free memory again falls below the minimum threshold.

In Figure 10.18, we see that at point a the amount of free memory drops below the minimum threshold, and the kernel begins reclaiming pages and adding them to the free-frame list. It continues until the maximum threshold is reached (point b). Over time, there are additional requests for memory, and at point c the amount of free memory again falls below the minimum threshold. Page reclamation resumes, only to be suspended when the amount of free memory reaches the maximum threshold (point d). This process continues as long as the system is running.

As mentioned above, the kernel reaper routine may adopt any page-replacement algorithm, but typically it uses some form of LRU approximation. Consider what may happen, though, if the reaper routine is unable to maintain the list of free frames below the minimum threshold. Under these circum-

stances, the reaper routine may begin to reclaim pages more aggressively. For example, perhaps it will suspend the second-chance algorithm and use pure FIFO. Another, more extreme, example occurs in Linux; when the amount of free memory falls to *very* low levels, a routine known as the **out-of-memory (OOM) killer** selects a process to terminate, thereby freeing its memory. How does Linux determine which process to terminate? Each process has what is known as an OOM score, with a higher score increasing the likelihood that the process could be terminated by the OOM killer routine. OOM scores are calculated according to the percentage of memory a process is using—the higher the percentage, the higher the OOM score. (OOM scores can be viewed in the /proc file system, where the score for a process with pid 2500 can be viewed as /proc/2500/oom_score.)

In general, not only can reaper routines vary how aggressively they reclaim memory, but the values of the minimum and maximum thresholds can be varied as well. These values can be set to default values, but some systems may allow a system administrator to configure them based on the amount of physical memory in the system.

10.5.4 Non-Uniform Memory Access

Thus far in our coverage of virtual memory, we have assumed that all main memory is created equal—or at least that it is accessed equally. On **non-uniform memory access (NUMA)** systems with multiple CPUs (Section 1.3.2), that is not the case. On these systems, a given CPU can access some sections of main memory faster than it can access others. These performance differences are caused by how CPUs and memory are interconnected in the system. Such a system is made up of multiple CPUs, each with its own local memory (Figure 10.19). The CPUs are organized using a shared system interconnect, and as you might expect, a CPU can access its local memory faster than memory local to another CPU. NUMA systems are without exception slower than systems in which all accesses to main memory are treated equally. However, as described in Section 1.3.2, NUMA systems can accommodate more CPUs and therefore achieve greater levels of throughput and parallelism.

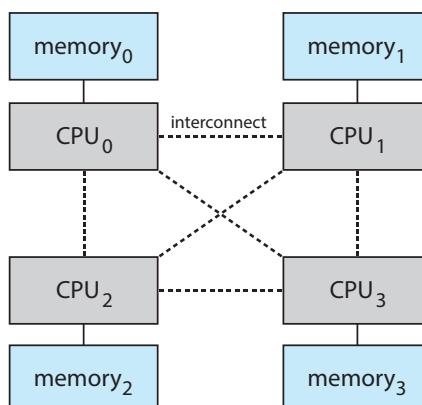


Figure 10.19 NUMA multiprocessing architecture.

Managing which page frames are stored at which locations can significantly affect performance in NUMA systems. If we treat memory as uniform in such a system, CPUs may wait significantly longer for memory access than if we modify memory allocation algorithms to take NUMA into account. We described some of these modifications in Section 5.5.4. Their goal is to have memory frames allocated “as close as possible” to the CPU on which the process is running. (The definition of *close* is “with minimum latency,” which typically means on the same system board as the CPU). Thus, when a process incurs a page fault, a NUMA-aware virtual memory system will allocate that process a frame as close as possible to the CPU on which the process is running.

To take NUMA into account, the scheduler must track the last CPU on which each process ran. If the scheduler tries to schedule each process onto its previous CPU, and the virtual memory system tries to allocate frames for the process close to the CPU on which it is being scheduled, then improved cache hits and decreased memory access times will result.

The picture is more complicated once threads are added. For example, a process with many running threads may end up with those threads scheduled on many different system boards. How should the memory be allocated in this case?

As we discussed in Section 5.7.1, Linux manages this situation by having the kernel identify a hierarchy of scheduling domains. The Linux CFS scheduler does not allow threads to migrate across different domains and thus incur memory access penalties. Linux also has a separate free-frame list for each NUMA node, thereby ensuring that a thread will be allocated memory from the node on which it is running. Solaris solves the problem similarly by creating **lgroups** (for “locality groups”) in the kernel. Each lgroup gathers together CPUs and memory, and each CPU in that group can access any memory in the group within a defined latency interval. In addition, there is a hierarchy of lgroups based on the amount of latency between the groups, similar to the hierarchy of scheduling domains in Linux. Solaris tries to schedule all threads of a process and allocate all memory of a process within an lgroup. If that is not possible, it picks nearby lgroups for the rest of the resources needed. This practice minimizes overall memory latency and maximizes CPU cache hit rates.

10.6 Thrashing

Consider what occurs if a process does not have “enough” frames—that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing. As you might expect, thrashing results in severe performance problems.

10.6.1 Cause of Thrashing

Consider the following scenario, which is based on the actual behavior of early paging systems. The operating system monitors CPU utilization. If CPU utiliza-

tion is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and *increases* the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page-fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in Figure 10.20, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm**). As mentioned earlier, local replacement requires that each process select from only its own set of allocated frames. Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase

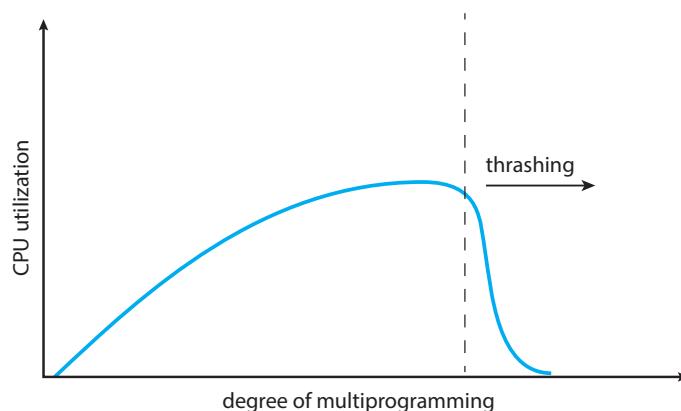


Figure 10.20 Thrashing.

because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it “needs”? One strategy starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A running program is generally composed of several different localities, which may overlap. For example, when a function is called, it defines a new locality. In this

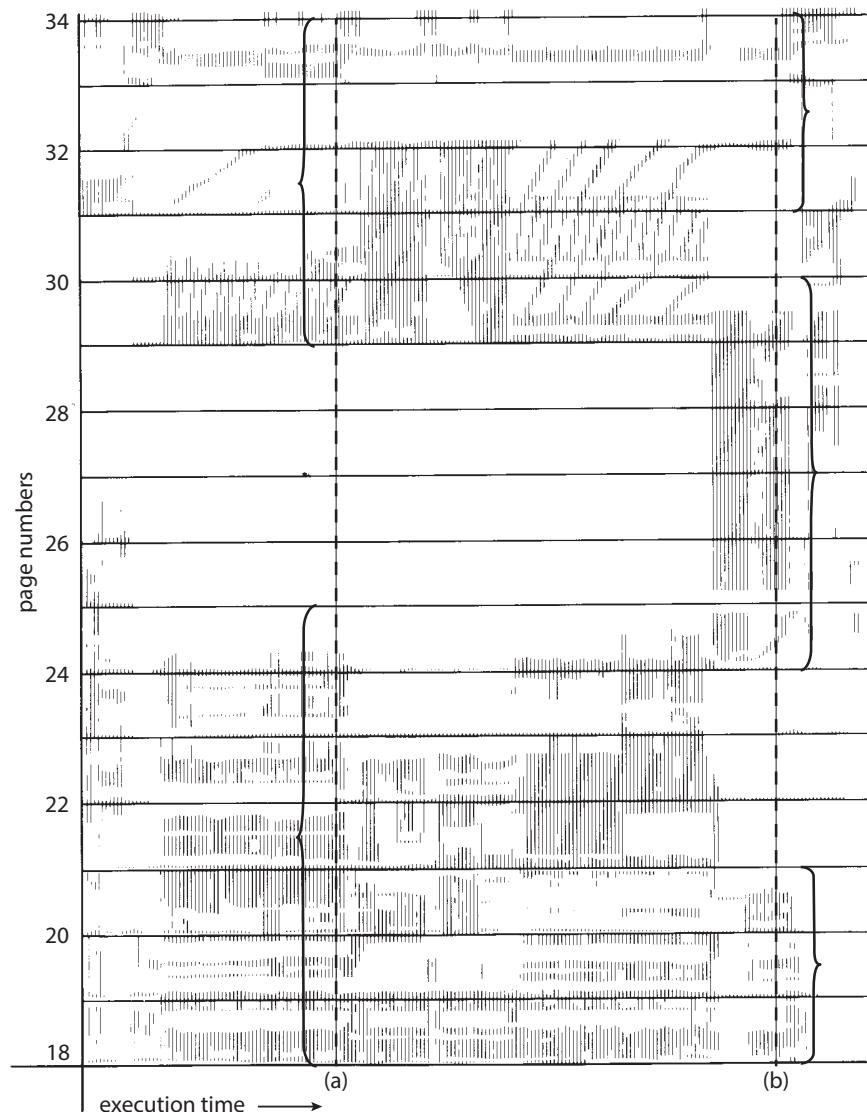


Figure 10.21 Locality in a memory-reference pattern.

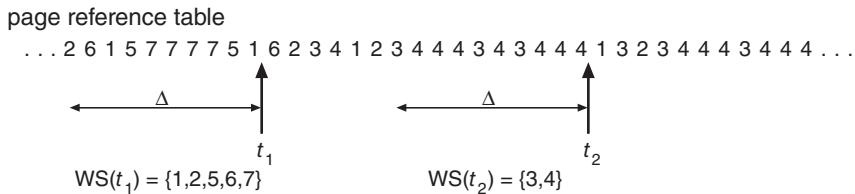


Figure 10.22 Working-set model.

locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.

Figure 10.21 illustrates the concept of locality and how a process's locality changes over time. At time (a), the locality is the set of pages $\{18, 19, 20, 21, 22, 23, 24, 29, 30, 33\}$. At time (b), the locality changes to $\{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33\}$. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.

Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

10.6.2 Working-Set Model

The **working-set model** is based on the assumption of locality. This model uses a parameter, Δ , to define the **working-set window**. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the **working set** (Figure 10.22). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure 10.22, if $\Delta = 10$ memory references, then the working set at time t_1 is $\{1, 2, 5, 6, 7\}$. By time t_2 , the working set has changed to $\{3, 4\}$.

The accuracy of the working set depends on the selection of Δ . If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities. In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set, then, is its size. If we compute the working-set size, WSS_i , for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

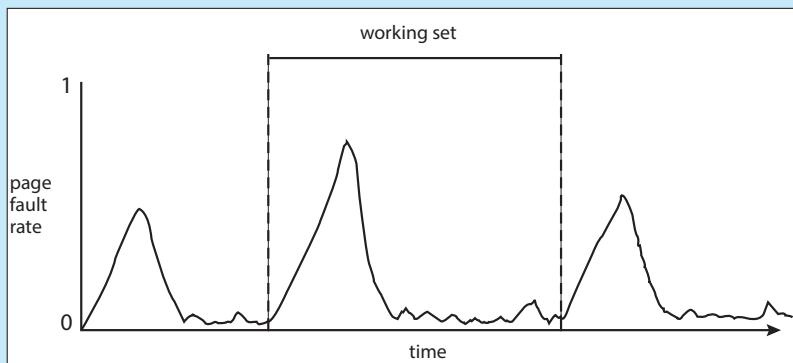
where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSS_i frames. If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

Once Δ has been selected, use of the working-set model is simple. The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The

WORKING SETS AND PAGE-FAULT RATES

There is a direct relationship between the working set of a process and its page-fault rate. Typically, as shown in Figure 10.22, the working set of a process changes over time as references to data and code sections move from one locality to another. Assuming there is sufficient memory to store the working set of a process (that is, the process is not thrashing), the page-fault rate of the process will transition between peaks and valleys over time. This general behavior is shown below:



A peak in the page-fault rate occurs when we begin demand-paging a new locality. However, once the working set of this new locality is in memory, the page-fault rate falls. When the process moves to a new working set, the page-fault rate rises toward a peak once again, returning to a lower rate once the new working set is loaded into memory. The span of time between the start of one peak and the start of the next peak represents the transition from one working set to another.

working-set window is a moving window. At each memory reference, a new reference appears at one end, and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.

We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit. For example, assume that Δ equals 10,000 references and that we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least one of these bits will be on. If it has not been used, these bits will be off. Pages with at least one bit on will be considered to be in the working set.

Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

10.6.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging (Section 10.9.1), but it seems a clumsy way to control thrashing. A strategy that uses the **page-fault frequency (PFF)** takes a more direct approach.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 10.23). If the actual page-fault rate exceeds the upper limit, we allocate the process

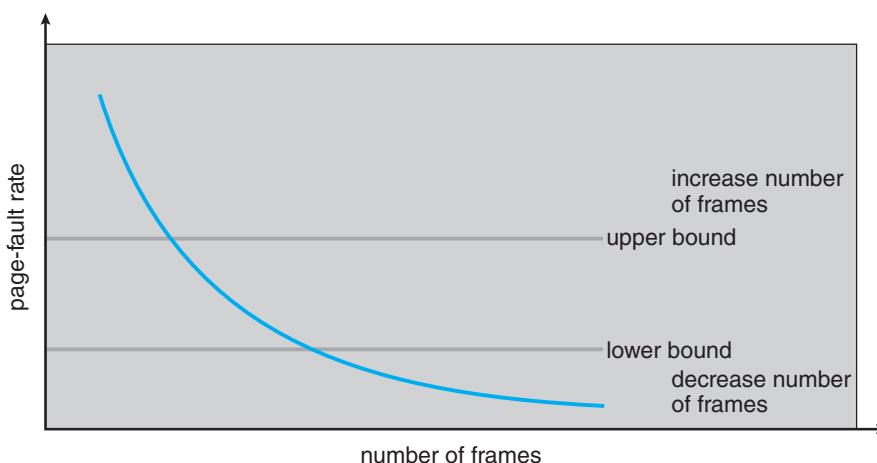


Figure 10.23 Page-fault frequency.

another frame. If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

As with the working-set strategy, we may have to swap out a process. If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store. The freed frames are then distributed to processes with high page-fault rates.

10.6.4 Current Practice

Practically speaking, thrashing and the resulting swapping have a disagreeably high impact on performance. The current best practice in implementing a computer system is to include enough physical memory, whenever possible, to avoid thrashing and swapping. From smartphones through large servers, providing enough memory to keep all working sets in memory concurrently, except under extreme conditions, provides the best user experience.

10.7 Memory Compression

An alternative to paging is **memory compression**. Here, rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

In Figure 10.24, the free-frame list contains six frames. Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames—15, 3, 35, and 26—to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames—say, three—and store their compressed versions in a single page frame.

In Figure 10.25, frame 7 is removed from the free-frame list. Frames 15, 3, and 35 are compressed and stored in frame 7, which is then stored in the list of compressed frames. The frames 15, 3, and 35 can now be moved to the free-frame list. If one of the three compressed frames is later referenced, a page fault occurs, and the compressed frame is decompressed, restoring the three pages 15, 3, and 35 in memory.

free-frame list



modified frame list



Figure 10.24 Free-frame list before compression.

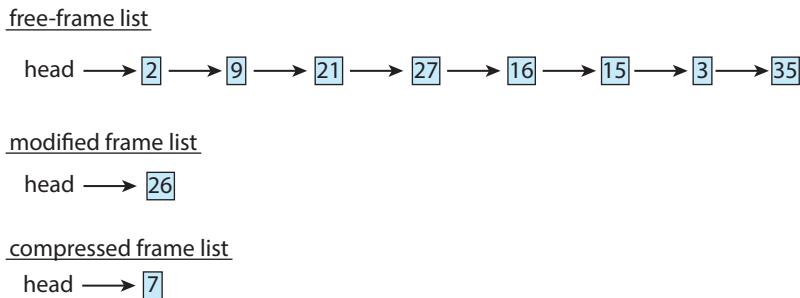


Figure 10.25 Free-frame list after compression

As we have noted, mobile systems generally do not support either standard swapping or swapping pages. Thus, memory compression is an integral part of the memory-management strategy for most mobile operating systems, including Android and iOS. In addition, both Windows 10 and macOS support memory compression. For Windows 10, Microsoft developed the [Universal Windows Platform \(UWP\)](#) architecture, which provides a common app platform for devices that run Windows 10, including mobile devices. UWP apps running on mobile devices are candidates for memory compression. macOS first supported memory compression with Version 10.9 of the operating system, first compressing LRU pages when free memory is short and then paging if that doesn't solve the problem. Performance tests indicate that memory compression is faster than paging even to SSD secondary storage on laptop and desktop macOS systems.

Although memory compression does require allocating free frames to hold the compressed pages, a significant memory saving can be realized, depending on the reductions achieved by the compression algorithm. (In the example above, the three frames were reduced to one-third of their original size.) As with any form of data compression, there is contention between the speed of the compression algorithm and the amount of reduction that can be achieved (known as the [compression ratio](#)). In general, higher compression ratios (greater reductions) can be achieved by slower, more computationally expensive algorithms. Most algorithms in use today balance these two factors, achieving relatively high compression ratios using fast algorithms. In addition, compression algorithms have improved by taking advantage of multiple computing cores and performing compression in parallel. For example, Microsoft's Xpress and Apple's WKdm compression algorithms are considered fast, and they report compressing pages to 30 to 50 percent of their original size.

10.8 Allocating Kernel Memory

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm such as those discussed in Section 10.4 and most likely contains free pages scattered throughout physical memory, as explained earlier. Remember, too, that if a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.
2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

In the following sections, we examine two strategies for managing free memory that is assigned to kernel processes: the “buddy system” and slab allocation.

10.8.1 Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Let’s consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies**—which we will call A_L and A_R —each 128 KB in size. One of these buddies is further divided into two 64-KB buddies— B_L and B_R . However, the next-highest power of 2 from 21 KB is 32 KB so either B_L or B_R is again divided into two 32-KB buddies, C_L and C_R . One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure 10.26, where C_L is the segment allocated to the 21-KB request.

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**. In Figure 10.26, for example, when the kernel releases the C_L unit it was allocated, the system can coalesce C_L and C_R into a 64-KB segment. This segment, B_L , can in turn be coalesced with its buddy B_R to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64-KB segment. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation. In the following section, we explore a memory allocation scheme where no space is lost due to fragmentation.

10.8.2 Slab Allocation

A second strategy for allocating kernel memory is known as **slab allocation**. A **slab** is made up of one or more physically contiguous pages. A **cache** consists

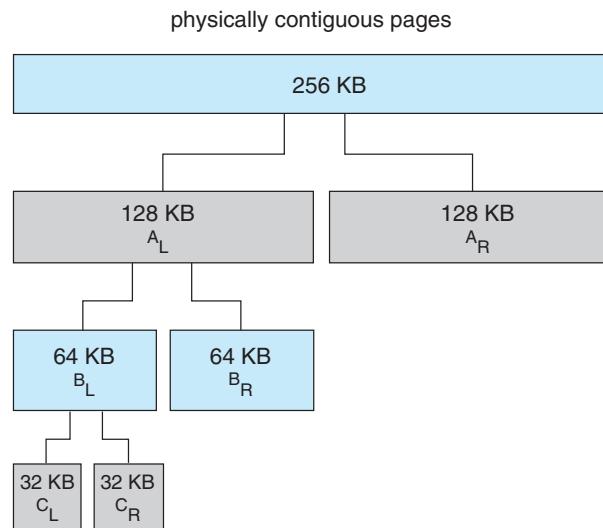


Figure 10.26 Buddy system allocation.

of one or more slabs. There is a single cache for each unique kernel data structure—for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth. The relationship among slabs, caches, and objects is shown in Figure 10.27. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

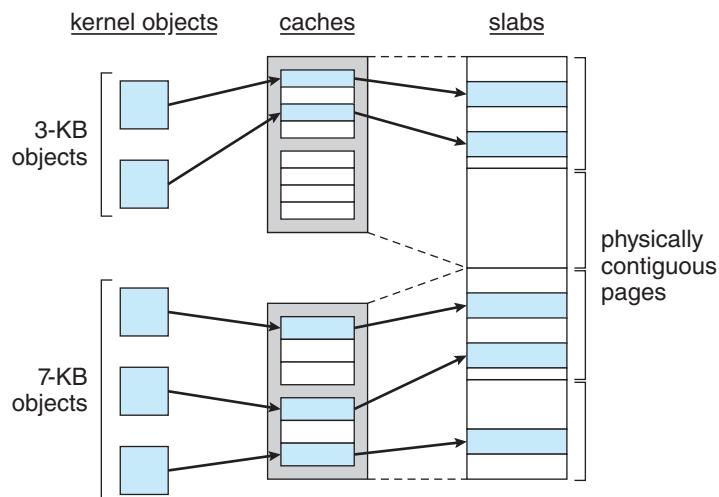


Figure 10.27 Slab allocation.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as `free`—are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type `struct task_struct`, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache. The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. **Full.** All objects in the slab are marked as used.
2. **Empty.** All objects in the slab are marked as free.
3. **Partial.** The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented. Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.
2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel. The act of allocating—and releasing—memory can be a time-consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

The slab allocator first appeared in the Solaris 2.4 kernel. Because of its general-purpose nature, this allocator is now also used for certain user-mode memory requests in Solaris. Linux originally used the buddy system; however, beginning with Version 2.2, the Linux kernel adopted the slab allocator. Linux

refers to its slab implementation as SLAB. Recent distributions of Linux include two other kernel memory allocators—the SLOB and SLUB allocators.

The SLOB allocator is designed for systems with a limited amount of memory, such as embedded systems. SLOB (which stands for “simple list of blocks”) maintains three lists of objects: *small* (for objects less than 256 bytes), *medium* (for objects less than 1,024 bytes), and *large* (for all other objects less than the size of a page). Memory requests are allocated from an object on the appropriate list using a first-fit policy.

Beginning with Version 2.6.24, the SLUB allocator replaced SLAB as the default allocator for the Linux kernel. SLUB reduced much of the overhead required by the SLAB allocator. For instance, whereas SLAB stores certain metadata with each slab, SLUB stores these data in the page structure the Linux kernel uses for each page. Additionally, SLUB does not include the per-CPU queues that the SLAB allocator maintains for objects in each cache. For systems with a large number of processors, the amount of memory allocated to these queues is significant. Thus, SLUB provides better performance as the number of processors on a system increases.

10.9 Other Considerations

The major decisions that we make for a paging system are the selections of a replacement algorithm and an allocation policy, which we discussed earlier in this chapter. There are many other considerations as well, and we discuss several of them here.

10.9.1 Prepaging

An obvious property of pure demand paging is the large number of page faults that occur when a process is started. This situation results from trying to get the initial locality into memory. **Prepaging** is an attempt to prevent this high level of initial paging. The strategy is to bring some—or all—of the pages that will be needed into memory at one time.

In a system using the working-set model, for example, we could keep with each process a list of the pages in its working set. If we must suspend a process (due to a lack of free frames), we remember the working set for that process. When the process is to be resumed (because I/O has finished or enough free frames have become available), we automatically bring back into memory its entire working set before restarting the process.

Prepaging may offer an advantage in some cases. The question is simply whether the cost of using prepaging is less than the cost of servicing the corresponding page faults. It may well be the case that many of the pages brought back into memory by prepaging will not be used.

Assume that s pages are prepaged and a fraction α of these s pages is actually used ($0 \leq \alpha \leq 1$). The question is whether the cost of the $s * \alpha$ saved page faults is greater or less than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages. If α is close to 0, prepaging loses; if α is close to 1, prepaging wins.

Note also that prepaging an executable program may be difficult, as it may be unclear exactly what pages should be brought in. Prepaging a file may be more predictable, since files are often accessed sequentially. The Linux

`readahead()` system call prefetches the contents of a file into memory so that subsequent accesses to the file will take place in main memory.

10.9.2 Page Size

The designers of an operating system for an existing machine seldom have a choice concerning the page size. However, when new machines are being designed, a decision regarding the best page size must be made. As you might expect, there is no single best page size. Rather, there is a set of factors that support various sizes. Page sizes are invariably powers of 2, generally ranging from 4,096 (2^{12}) to 4,194,304 (2^{22}) bytes.

How do we select a page size? One concern is the size of the page table. For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table. For a virtual memory of 4 MB (2^{22}), for example, there would be 4,096 pages of 1,024 bytes but only 512 pages of 8,192 bytes. Because each active process must have its own copy of the page table, a large page size is desirable.

Memory is better utilized with smaller pages, however. If a process is allocated memory starting at location 00000 and continuing until it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated (because pages are the units of allocation) but will be unused (creating internal fragmentation). Assuming independence of process size and page size, we can expect that, on the average, half of the final page of each process will be wasted. This loss is only 256 bytes for a page of 512 bytes but is 4,096 bytes for a page of 8,192 bytes. To minimize internal fragmentation, then, we need a small page size.

Another problem is the time required to read or write a page. As you will see in Section 11.1, when the storage device is an HDD, I/O time is composed of seek, latency, and transfer times. Transfer time is proportional to the amount transferred (that is, the page size)—a fact that would seem to argue for a small page size. However, latency and seek time normally dwarf transfer time. At a transfer rate of 50 MB per second, it takes only 0.01 milliseconds to transfer 512 bytes. Latency time, though, is perhaps 3 milliseconds, and seek time 5 milliseconds. Of the total I/O time (8.01 milliseconds), therefore, only about 0.1 percent is attributable to the actual transfer. Doubling the page size increases I/O time to only 8.02 milliseconds. It takes 8.02 milliseconds to read a single page of 1,024 bytes but 16.02 milliseconds to read the same amount as two pages of 512 bytes each. Thus, a desire to minimize I/O time argues for a larger page size.

With a smaller page size, though, total I/O should be reduced, since locality will be improved. A smaller page size allows each page to match program locality more accurately. For example, consider a process 200 KB in size, of which only half (100 KB) is actually used in an execution. If we have only one large page, we must bring in the entire page, a total of 200 KB transferred and allocated. If instead we had pages of only 1 byte, then we could bring in only the 100 KB that are actually used, resulting in only 100 KB transferred and allocated. With a smaller page size, then, we have better **resolution**, allowing us to isolate only the memory that is actually needed. With a larger page size, we must allocate and transfer not only what is needed but also anything else

that happens to be in the page, whether it is needed or not. Thus, a smaller page size should result in less I/O and less total allocated memory.

But did you notice that with a page size of 1 byte, we would have a page fault for *each* byte? A process of 200 KB that used only half of that memory would generate only one page fault with a page size of 200 KB but 102,400 page faults with a page size of 1 byte. Each page fault generates the large amount of overhead needed for processing the interrupt, saving registers, replacing a page, queuing for the paging device, and updating tables. To minimize the number of page faults, we need to have a large page size.

Other factors must be considered as well (such as the relationship between page size and sector size on the paging device). The problem has no best answer. As we have seen, some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. Nevertheless, the historical trend is toward larger page sizes, even for mobile systems. Indeed, the first edition of *Operating System Concepts* (1983) used 4,096 bytes as the upper bound on page sizes, and this value was the most common page size in 1990. Modern systems may now use much larger page sizes, as you will see in the following section.

10.9.3 TLB Reach

In Chapter 9, we introduced the **hit ratio** of the TLB. Recall that the hit ratio for the TLB refers to the percentage of virtual address translations that are resolved in the TLB rather than the page table. Clearly, the hit ratio is related to the number of entries in the TLB, and the way to increase the hit ratio is by increasing the number of entries. This, however, does not come cheaply, as the associative memory used to construct the TLB is both expensive and power hungry.

Related to the hit ratio is a similar metric: the **TLB reach**. The TLB reach refers to the amount of memory accessible from the TLB and is simply the number of entries multiplied by the page size. Ideally, the working set for a process is stored in the TLB. If it is not, the process will spend a considerable amount of time resolving memory references in the page table rather than the TLB. If we double the number of entries in the TLB, we double the TLB reach. However, for some memory-intensive applications, this may still prove insufficient for storing the working set.

Another approach for increasing the TLB reach is to either increase the size of the page or provide multiple page sizes. If we increase the page size—say, from 4 KB to 16 KB—we quadruple the TLB reach. However, this may lead to an increase in fragmentation for some applications that do not require such a large page size. Alternatively, most architectures provide support for more than one page size, and an operating system can be configured to take advantage of this support. For example, the default page size on Linux systems is 4 KB; however, Linux also provides **huge pages**, a feature that designates a region of physical memory where larger pages (for example, 2 MB) may be used.

Recall from Section 9.7 that the ARMv8 architecture provides support for pages and regions of different sizes. Additionally, each TLB entry in the ARMv8 contains a **contiguous bit**. If this bit is set for a particular TLB entry, that entry maps contiguous (adjacent) blocks of memory. Three possible arrangements of

contiguous blocks can be mapped in a single TLB entry, thereby increasing the TLB reach:

1. 64-KB TLB entry comprising 16×4 KB adjacent blocks.
2. 1-GB TLB entry comprising 32×32 MB adjacent blocks.
3. 2-MB TLB entry comprising either 32×64 KB adjacent blocks, or 128×16 KB adjacent blocks.

Providing support for multiple page sizes may require the operating system—rather than hardware—to manage the TLB. For example, one of the fields in a TLB entry must indicate the size of the page frame corresponding to the entry—or, in the case of ARM architectures, must indicate that the entry refers to a contiguous block of memory. Managing the TLB in software and not hardware comes at a cost in performance. However, the increased hit ratio and TLB reach offset the performance costs.

10.9.4 Inverted Page Tables

Section 9.4.3 introduced the concept of the inverted page table. The purpose of this form of page management is to reduce the amount of physical memory needed to track virtual-to-physical address translations. We accomplish this savings by creating a table that has one entry per page of physical memory, indexed by the pair <process-id, page-number>.

Because they keep information about which virtual memory page is stored in each physical frame, inverted page tables reduce the amount of physical memory needed to store this information. However, the inverted page table no longer contains complete information about the logical address space of a process, and that information is required if a referenced page is not currently in memory. Demand paging requires this information to process page faults. For the information to be available, an external page table (one per process) must be kept. Each such table looks like the traditional per-process page table and contains information on where each virtual page is located.

But do external page tables negate the utility of inverted page tables? Since these tables are referenced only when a page fault occurs, they do not need to be available quickly. Instead, they are themselves paged in and out of memory as necessary. Unfortunately, a page fault may now cause the virtual memory manager to generate another page fault as it pages in the external page table it needs to locate the virtual page on the backing store. This special case requires careful handling in the kernel and a delay in the page-lookup processing.

10.9.5 Program Structure

Demand paging is designed to be transparent to the user program. In many cases, the user is completely unaware of the paged nature of memory. In other cases, however, system performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.

Let's look at a contrived but informative example. Assume that pages are 128 words in size. Consider a C program whose function is to initialize to 0 each element of a 128-by-128 array. The following code is typical:

```

int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;

```

Notice that the array is stored row major; that is, the array is stored $\text{data}[0][0]$, $\text{data}[0][1]$, ..., $\text{data}[0][127]$, $\text{data}[1][0]$, $\text{data}[1][1]$, ..., $\text{data}[127][127]$. For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates fewer than 128 frames to the entire program, then its execution will result in $128 \times 128 = 16,384$ page faults.

In contrast, suppose we change the code to

```

int i, j;
int[128][128] data;

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;

```

This code zeros all the words on one page before starting the next page, reducing the number of page faults to 128.

Careful selection of data structures and programming structures can increase locality and hence lower the page-fault rate and the number of pages in the working set. For example, a stack has good locality, since access is always made to the top. A hash table, in contrast, is designed to scatter references, producing bad locality. Of course, locality of reference is just one measure of the efficiency of the use of a data structure. Other heavily weighted factors include search speed, total number of memory references, and total number of pages touched.

At a later stage, the compiler and loader can have a significant effect on paging. Separating code and data and generating reentrant code means that code pages can be read-only and hence will never be modified. Clean pages do not have to be paged out to be replaced. The loader can avoid placing routines across page boundaries, keeping each routine completely in one page. Routines that call each other many times can be packed into the same page. This packaging is a variant of the bin-packing problem of operations research: try to pack the variable-sized load segments into the fixed-sized pages so that interpage references are minimized. Such an approach is particularly useful for large page sizes.

10.9.6 I/O Interlock and Page Locking

When demand paging is used, we sometimes need to allow some of the pages to be **locked** in memory. One such situation occurs when I/O is done to or from user (virtual) memory. I/O is often implemented by a separate I/O processor. For example, a controller for a USB storage device is generally given the number

of bytes to transfer and a memory address for the buffer (Figure 10.28). When the transfer is complete, the CPU is interrupted.

We must be sure the following sequence of events does not occur: A process issues an I/O request and is put in a queue for that I/O device. Meanwhile, the CPU is given to other processes. These processes cause page faults, and one of them, using a global replacement algorithm, replaces the page containing the memory buffer for the waiting process. The pages are paged out. Some time later, when the I/O request advances to the head of the device queue, the I/O occurs to the specified address. However, this frame is now being used for a different page belonging to another process.

There are two common solutions to this problem. One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory. I/O takes place only between system memory and the I/O device. Thus, to write a block on tape, we first copy the block to system memory and then write it to tape. This extra copying may result in unacceptably high overhead.

Another solution is to allow pages to be locked into memory. Here, a lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement. Under this approach, to write a block to disk, we lock into memory the pages containing the block. The system can then continue as usual. Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

Lock bits are used in various situations. Frequently, some or all of the operating-system kernel is locked into memory. Many operating systems cannot tolerate a page fault caused by the kernel or by a specific kernel module, including the one performing memory management. User processes may also need to lock pages into memory. A database process may want to manage a chunk of memory, for example, moving blocks between secondary storage and

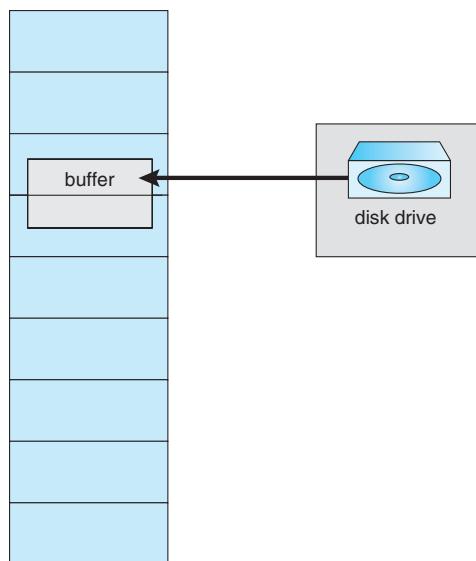


Figure 10.28 The reason why frames used for I/O must be in memory.

memory itself because it has the best knowledge of how it is going to use its data. Such **pinning** of pages in memory is fairly common, and most operating systems have a system call allowing an application to request that a region of its logical address space be pinned. Note that this feature could be abused and could cause stress on the memory-management algorithms. Therefore, an application frequently requires special privileges to make such a request.

Another use for a lock bit involves normal page replacement. Consider the following sequence of events: A low-priority process faults. Selecting a replacement frame, the paging system reads the necessary page into memory. Ready to continue, the low-priority process enters the ready queue and waits for the CPU. Since it is a low-priority process, it may not be selected by the CPU scheduler for a time. While the low-priority process waits, a high-priority process faults. Looking for a replacement, the paging system sees a page that is in memory but has not been referenced or modified: it is the page that the low-priority process just brought in. This page looks like a perfect replacement. It is clean and will not need to be written out, and it apparently has not been used for a long time.

Whether the high-priority process should be able to replace the low-priority process is a policy decision. After all, we are simply delaying the low-priority process for the benefit of the high-priority process. However, we are wasting the effort spent to bring in the page for the low-priority process. If we decide to prevent replacement of a newly brought-in page until it can be used at least once, then we can use the lock bit to implement this mechanism. When a page is selected for replacement, its lock bit is turned on. It remains on until the faulting process is again dispatched.

Using a lock bit can be dangerous: the lock bit may get turned on but never turned off. Should this situation occur (because of a bug in the operating system, for example), the locked frame becomes unusable. For instance, Solaris allows locking “hints,” but it is free to disregard these hints if the free-frame pool becomes too small or if an individual process requests that too many pages be locked in memory.

10.10 Operating-System Examples

In this section, we describe how Linux, Windows and Solaris manage virtual memory.

10.10.1 Linux

In Section 10.8.2, we discussed how Linux manages kernel memory using slab allocation. We now cover how Linux manages virtual memory. Linux uses demand paging, allocating pages from a list of free frames. In addition, it uses a global page-replacement policy similar to the LRU-approximation clock algorithm described in Section 10.4.5.2. To manage memory, Linux maintains two types of page lists: an `active_list` and an `inactive_list`. The `active_list` contains the pages that are considered in use, while the `inactive_list` contains pages that have not recently been referenced and are eligible to be reclaimed.

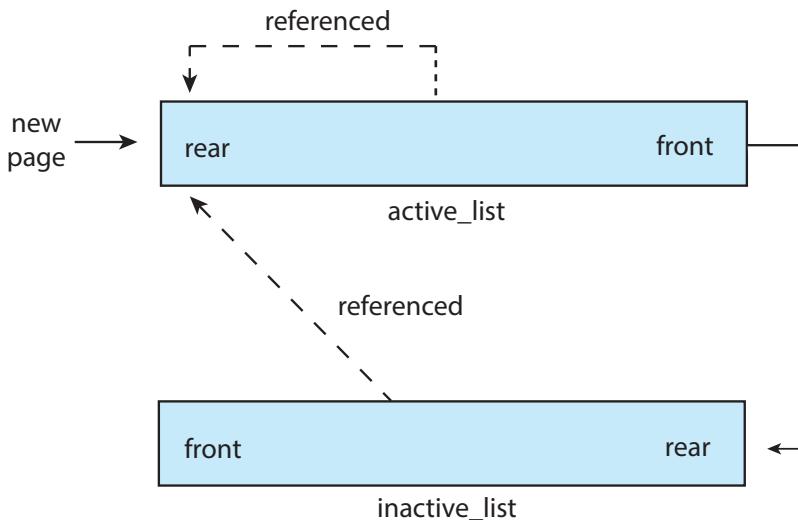


Figure 10.29 The Linux `active_list` and `inactive_list` structures.

Each page has an *accessed* bit that is set whenever the page is referenced. (The actual bits used to mark page access vary by architecture.) When a page is first allocated, its accessed bit is set, and it is added to the rear of the `active_list`. Similarly, whenever a page in the `active_list` is referenced, its accessed bit is set, and the page moves to the rear of the list. Periodically, the accessed bits for pages in the `active_list` are reset. Over time, the least recently used page will be at the front of the `active_list`. From there, it may migrate to the rear of the `inactive_list`. If a page in the `inactive_list` is referenced, it moves back to the rear of the `active_list`. This pattern is illustrated in Figure 10.29.

The two lists are kept in relative balance, and when the `active_list` grows much larger than the `inactive_list`, pages at the front of the `active_list` move to the `inactive_list`, where they become eligible for reclamation. The Linux kernel has a page-out daemon process `kswapd` that periodically awakens and checks the amount of free memory in the system. If free memory falls below a certain threshold, `kswapd` begins scanning pages in the `inactive_list` and reclaiming them for the free list. Linux virtual memory management is discussed in greater detail in Chapter 20.

10.10.2 Windows

Windows 10 supports 32- and 64-bit systems running on Intel (IA-32 and x86-64) and ARM architectures. On 32-bit systems, the default virtual address space of a process is 2 GB, although it can be extended to 3 GB. 32-bit systems support 4 GB of physical memory. On 64-bit systems, Windows 10 has a 128-TB virtual address space and supports up to 24 TB of physical memory. (Versions of Windows Server support up to 128 TB of physical memory.) Windows 10 implements most of the memory-management features described thus far, including shared libraries, demand paging, copy-on-write, paging, and memory compression.

Windows 10 implements virtual memory using demand paging with **clustering**, a strategy that recognizes locality of memory references and therefore handles page faults by bringing in not only the faulting page but also several pages immediately preceding and following the faulting page. The size of a cluster varies by page type. For a data page, a cluster contains three pages (the page before and the page after the faulting page); all other page faults have a cluster size of seven.

A key component of virtual memory management in Windows 10 is working-set management. When a process is created, it is assigned a working-set minimum of 50 pages and a working-set maximum of 345 pages. The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory; if sufficient memory is available, a process may be assigned as many pages as its **working-set maximum**. Unless a process is configured with **hard working-set limits**, these values may be ignored. A process can grow beyond its working-set maximum if sufficient memory is available. Similarly, the amount of memory allocated to a process can shrink below the minimum in periods of high demand for memory.

Windows uses the LRU-approximation clock algorithm, as described in Section 10.4.5.2, with a combination of local and global page-replacement policies. The virtual memory manager maintains a list of free page frames. Associated with this list is a threshold value that indicates whether sufficient free memory is available. If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from the list of free pages. If a process that is at its working-set maximum incurs a page fault and sufficient memory is available, the process is allocated a free page, which allows it to grow beyond its working-set maximum. If the amount of free memory is insufficient, however, the kernel must select a page from the process's working set for replacement using a local LRU page-replacement policy.

When the amount of free memory falls below the threshold, the virtual memory manager uses a global replacement tactic known as **automatic working-set trimming** to restore the value to a level above the threshold. Automatic working-set trimming works by evaluating the number of pages allocated to processes. If a process has been allocated more pages than its working-set minimum, the virtual memory manager removes pages from the working set until either there is sufficient memory available or the process has reached its working-set minimum. Larger processes that have been idle are targeted before smaller, active processes. The trimming procedure continues until there is sufficient free memory, even if it is necessary to remove pages from a process already below its working set minimum. Windows performs working-set trimming on both user-mode and system processes.

10.10.3 Solaris

In Solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains. Therefore, it is imperative that the kernel keep a sufficient amount of free memory available. Associated with this list of free pages is a parameter—`lotsfree`—that represents a threshold to begin paging. The `lotsfree` parameter is typically set to 1/64 the size of the physical memory. Four times per second, the kernel checks whether the amount of free memory is less than `lotsfree`. If the number of

free pages falls below `lotsfree`, a process known as a **pageout** starts up. The pageout process is similar to the second-chance algorithm described in Section 10.4.5.2, except that it uses two hands while scanning pages, rather than one.

The pageout process works as follows: The front hand of the clock scans all pages in memory, setting the reference bit to 0. Later, the back hand of the clock examines the reference bit for the pages in memory, appending each page whose reference bit is still set to 0 to the free list and writing its contents to secondary storage if it has been modified. Solaris also manages minor page faults by allowing a process to reclaim a page from the free list if the page is accessed before being reassigned to another process.

The pageout algorithm uses several parameters to control the rate at which pages are scanned (known as the `scanrate`). The scanrate is expressed in pages per second and ranges from `slowscan` to `fastscan`. When free memory falls below `lotsfree`, scanning occurs at `slowscan` pages per second and progresses to `fastscan`, depending on the amount of free memory available. The default value of `slowscan` is 100 pages per second. `Fastscan` is typically set to the value (total physical pages)/2 pages per second, with a maximum of 8,192 pages per second. This is shown in Figure 10.30 (with `fastscan` set to the maximum).

The distance (in pages) between the hands of the clock is determined by a system parameter, `handspread`. The amount of time between the front hand's clearing a bit and the back hand's investigating its value depends on the `scanrate` and the `handspread`. If `scanrate` is 100 pages per second and `handspread` is 1,024 pages, 10 seconds can pass between the time a bit is set by the front hand and the time it is checked by the back hand. However, because of the demands placed on the memory system, a `scanrate` of several thousand is not uncommon. This means that the amount of time between clearing and investigating a bit is often a few seconds.

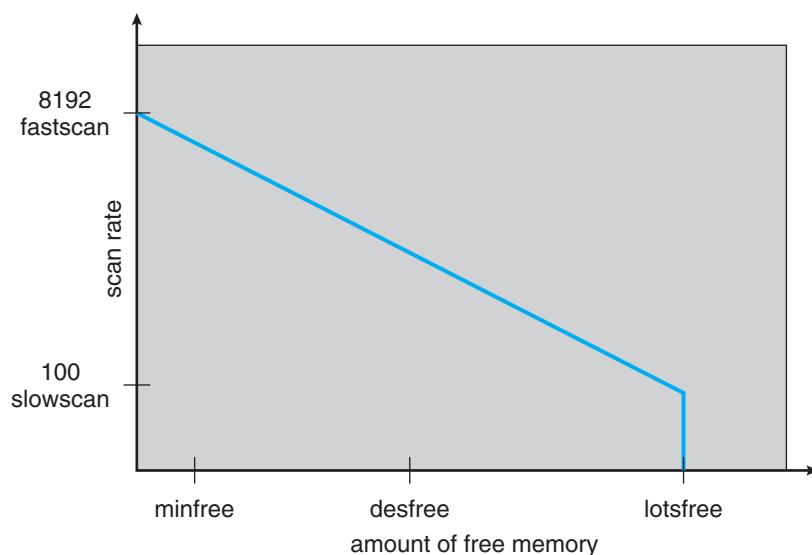


Figure 10.30 Solaris page scanner.

As mentioned above, the pageout process checks memory four times per second. However, if free memory falls below the value of `desfree` (the desired amount of free memory in the system), pageout will run a hundred times per second with the intention of keeping at least `desfree` free memory available (Figure 10.30). If the pageout process is unable to keep the amount of free memory at `desfree` for a 30-second average, the kernel begins swapping processes, thereby freeing all pages allocated to swapped processes. In general, the kernel looks for processes that have been idle for long periods of time. If the system is unable to maintain the amount of free memory at `minfree`, the pageout process is called for every request for a new page.

The page-scanning algorithm skips pages belonging to libraries that are being shared by several processes, even if they are eligible to be claimed by the scanner. The algorithm also distinguishes between pages allocated to processes and pages allocated to regular data files. This is known as **priority paging** and is covered in Section 14.6.2.

10.11 Summary

- Virtual memory abstracts physical memory into an extremely large uniform array of storage.
- The benefits of virtual memory include the following: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- Demand paging is a technique whereby pages are loaded only when they are demanded during program execution. Pages that are never demanded are thus never loaded into memory.
- A page fault occurs when a page that is currently not in memory is accessed. The page must be brought from the backing store into an available page frame in memory.
- Copy-on-write allows a child process to share the same address space as its parent. If either the child or the parent process writes (modifies) a page, a copy of the page is made.
- When available memory runs low, a page-replacement algorithm selects an existing page in memory to replace with a new page. Page-replacement algorithms include FIFO, optimal, and LRU. Pure LRU algorithms are impractical to implement, and most systems instead use LRU-approximation algorithms.
- Global page-replacement algorithms select a page from any process in the system for replacement, while local page-replacement algorithms select a page from the faulting process.
- Thrashing occurs when a system spends more time paging than executing.
- A locality represents a set of pages that are actively used together. As a process executes, it moves from locality to locality. A working set is based on locality and is defined as the set of pages currently in use by a process.

- Memory compression is a memory-management technique that compresses a number of pages into a single page. Compressed memory is an alternative to paging and is used on mobile systems that do not support paging.
- Kernel memory is allocated differently than user-mode processes; it is allocated in contiguous chunks of varying sizes. Two common techniques for allocating kernel memory are (1) the buddy system and (2) slab allocation.
- TLB reach refers to the amount of memory accessible from the TLB and is equal to the number of entries in the TLB multiplied by the page size. One technique for increasing TLB reach is to increase the size of pages.
- Linux, Windows, and Solaris manage virtual memory similarly, using demand paging and copy-on-write, among other features. Each system also uses a variation of LRU approximation known as the clock algorithm.

Practice Exercises

- 10.1** Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.
- 10.2** Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p , and n distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:
- a. What is a lower bound on the number of page faults?
 - b. What is an upper bound on the number of page faults?
- 10.3** Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from “bad” to “perfect” according to their page-fault rate. Separate those algorithms that suffer from Belady’s anomaly from those that do not.
- a. LRU replacement
 - b. FIFO replacement
 - c. Optimal replacement
 - d. Second-chance replacement
- 10.4** An operating system supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfers 1 million words per second. The following statistical measurements were obtained from the system:
- One percent of all instructions executed accessed a page other than the current page.
 - Of the instructions that accessed another page, 80 percent accessed a page already in memory.

- When a new page was required, the replaced page was modified 50 percent of the time.

Calculate the effective instruction time on this system, assuming that the system is running one process only and that the processor is idle during drum transfers.

- 10.5** Consider the page table for a system with 12-bit virtual and physical addresses and 256-byte pages.

Page	Page Frame
0	-
1	2
2	C
3	A
4	-
5	4
6	3
7	-
8	B
9	0

The list of free page frames is D, E, F (that is, D is at the head of the list, E is second, and F is last). A dash for a page frame indicates that the page is not in memory.

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal.

- 9EF
- 111
- 700
- 0FF

- 10.6** Discuss the hardware functions required to support demand paging.

- 10.7** Consider the two-dimensional array A:

```
int A[] [] = new int[100][100];
```

where $A[0][0]$ is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops? Use LRU replacement, and assume

that page frame 1 contains the process and the other two are initially empty.

- a.

```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```
- b.

```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

10.8 Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

10.9 Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

- 10.10** Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware, or explain why it is not possible to do so. If it is possible, calculate what the cost would be.
- 10.11** You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.
- 10.12** Segmentation is similar to paging but uses variable-sized “pages.” Define two segment-replacement algorithms, one based on the FIFO page-replacement scheme and the other on the LRU page-replacement scheme. Remember that since segments are not the same size, the segment that is chosen for replacement may be too small to leave enough

consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated and strategies for systems where they can.

- 10.13** Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of the CPU and the paging disk. Three alternative results are shown below. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?
- CPU utilization 13 percent; disk utilization 97 percent
 - CPU utilization 87 percent; disk utilization 3 percent
 - CPU utilization 13 percent; disk utilization 3 percent
- 10.14** We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can the page table be set up to simulate base and limit registers? How can it be, or why can it not be?

Further Reading

The working-set model was developed by [Denning (1968)]. The enhanced clock algorithm is discussed by [Carr and Hennessy (1981)]. [Russinovich et al. (2017)] describe how Windows implements virtual memory and memory compression. Compressed memory in Windows 10 is further discussed in <http://www.makeuseof.com/tag/ram-compression-improves-memory-responsiveness-windows-10>.

[McDougall and Mauro (2007)] discuss virtual memory in Solaris. Virtual memory techniques in Linux are described in [Love (2010)] and [Mauerer (2008)]. FreeBSD is described in [McKusick et al. (2015)].

Bibliography

- [Carr and Hennessy (1981)]** W. R. Carr and J. L. Hennessy, “WSClock—A Simple and Effective Algorithm for Virtual Memory Management”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1981), pages 87–95.
- [Denning (1968)]** P. J. Denning, “The Working Set Model for Program Behavior”, *Communications of the ACM*, Volume 11, Number 5 (1968), pages 323–333.
- [Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).
- [Mauerer (2008)]** W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).
- [McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

[McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System—Second Edition*, Pearson (2015).

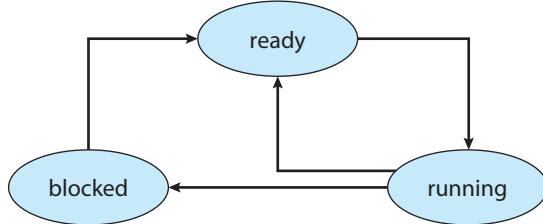
[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).

Chapter 10 Exercises

10.15 Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

- TLB miss with no page fault
- TLB miss with page fault
- TLB hit with no page fault
- TLB hit with page fault

10.16 A simplified view of thread states is *ready*, *running*, and *blocked*, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O).



Assuming a thread is in the *running* state, answer the following questions, and explain your answers:

- Will the thread change state if it incurs a page fault? If so, to what state will it change?
- Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?
- Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?

10.17 Consider a system that uses pure demand paging.

- When a process first starts execution, how would you characterize the page-fault rate?
- Once the working set for a process is loaded into memory, how would you characterize the page-fault rate?
- Assume that a process changes its locality and the size of the new working set is too large to be stored in available free memory. Identify some options system designers could choose from to handle this situation.

10.18 The following is a page table for a system with 12-bit virtual and physical addresses and 256-byte pages. Free page frames are to be allocated in the order 9, F, D. A dash for a page frame indicates that the page is not in memory.

Page	Page Frame
0	0 x 4
1	0 x B
2	0 x A
3	-
4	-
5	0 x 2
6	-
7	0 x 0
8	0 x C
9	0 x 1

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. In the case of a page fault, you must use one of the free frames to update the page table and resolve the logical address to its corresponding physical address.

- 0x2A1
- 0x4E6
- 0x94A
- 0x316

- 10.19** What is the copy-on-write feature, and under what circumstances is its use beneficial? What hardware support is required to implement this feature?
- 10.20** A certain computer provides its users with a virtual memory space of 2^{32} bytes. The computer has 2^{22} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.
- 10.21** Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.
- Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?
- 10.22** Consider the page table for a system with 16-bit virtual and physical addresses and 4,096-byte pages.

Page	Page Frame	Reference Bit
0	9	0
1	-	0
2	10	0
3	15	0
4	6	0
5	13	0
6	8	0
7	12	0
8	7	0
9	-	0
10	5	0
11	4	0
12	1	0
13	0	0
14	-	0
15	2	0

The reference bit for a page is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates that the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

- Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.
 - 0x621C
 - 0xF0A3
 - 0xBC1A
 - 0x5BAA
 - 0x0BA1
 - Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.
 - From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?
- 10.23** When a page fault occurs, the process requesting the page must block while waiting for the page to be brought from disk into physical memory. Assume that there exists a process with five user-level threads and that the mapping of user threads to kernel threads is many to one. If

one user thread incurs a page fault while accessing its stack, would the other user threads belonging to the same process also be affected by the page fault—that is, would they also have to wait for the faulting page to be brought into memory? Explain.

10.24 Apply the (1) FIFO, (2) LRU, and (3) optimal (OPT) replacement algorithms for the following page-reference strings:

- 2,6,9,2,4,2,1,7,3,0,5,2,1,2,9,5,7,3,8,5
- 0,6,3,0,2,6,3,5,2,4,1,3,0,6,1,4,2,3,5,7
- 3,1,4,2,5,4,1,3,5,2,0,1,1,0,2,3,4,5,0,1
- 4,2,1,7,9,8,3,5,2,6,8,1,0,7,2,4,1,3,5,8
- 0,1,2,3,4,4,3,2,1,0,0,1,2,3,4,4,3,2,1,0

Indicate the number of page faults for each algorithm assuming demand paging with three frames.

10.25 Assume that you are monitoring the rate at which the pointer in the clock algorithm moves. (The pointer indicates the candidate page for replacement.) What can you say about the system if you notice the following behavior:

- a. Pointer is moving fast.
- b. Pointer is moving slow.

10.26 Discuss situations in which the least frequently used (LFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

10.27 Discuss situations in which the most frequently used (MFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

10.28 The KHIE (pronounced “k-hi”) operating system uses a FIFO replacement algorithm for resident pages and a free-frame pool of recently used pages. Assume that the free-frame pool is managed using the LRU replacement policy. Answer the following questions:

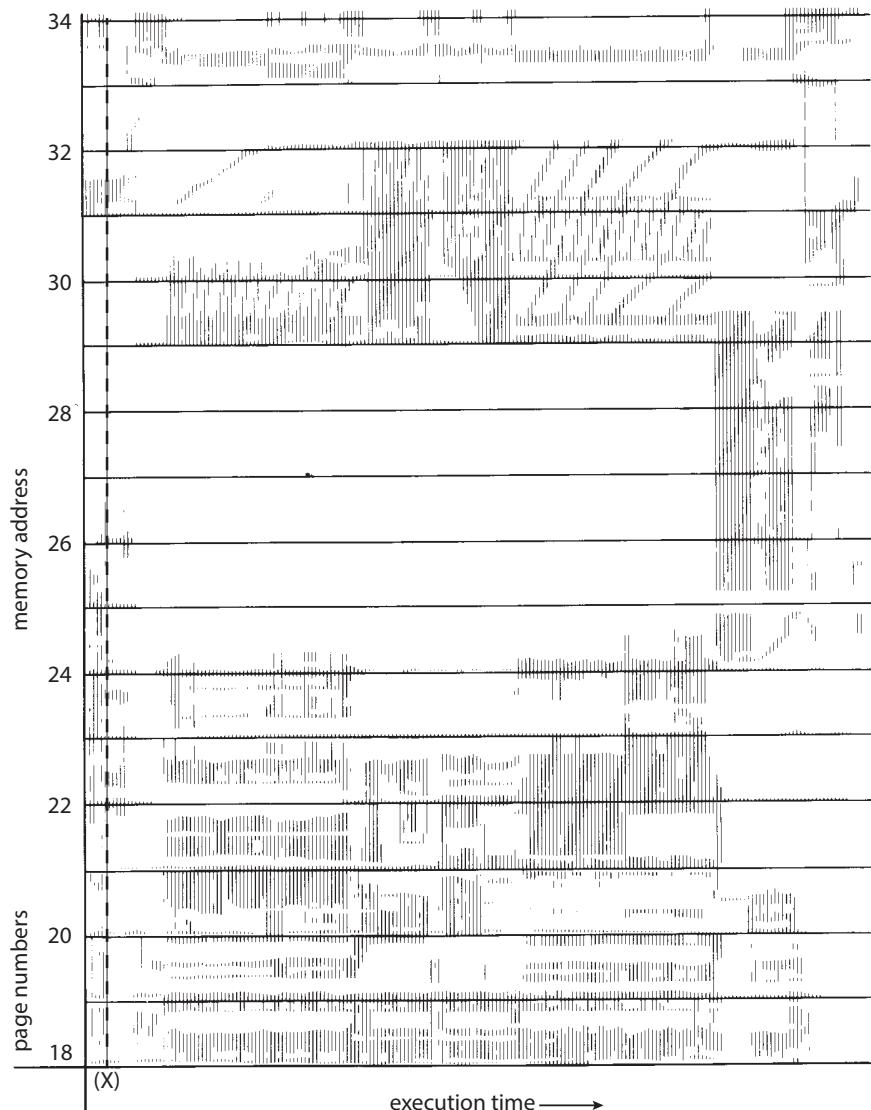
- a. If a page fault occurs and the page does not exist in the free-frame pool, how is free space generated for the newly requested page?
- b. If a page fault occurs and the page exists in the free-frame pool, how are the resident page set and the free-frame pool managed to make space for the requested page?
- c. To what does the system degenerate if the number of resident pages is set to one?
- d. To what does the system degenerate if the number of pages in the free-frame pool is zero?

- 10.29** Consider a demand-paging system with the following time-measured utilizations:

CPU utilization	20%
Paging disk	97.7%
Other I/O devices	5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization. Explain your answers.

- a. Install a faster CPU.
 - b. Install a bigger paging disk.
 - c. Increase the degree of multiprogramming.
 - d. Decrease the degree of multiprogramming.
 - e. Install more main memory.
 - f. Install a faster hard disk or multiple controllers with multiple hard disks.
 - g. Add prepaging to the page-fetch algorithms.
 - h. Increase the page size.
- 10.30** Explain why minor page faults take less time to resolve than major page faults.
- 10.31** Explain why compressed memory is used in operating systems for mobile devices.
- 10.32** Suppose that a machine provides instructions that can access memory locations using the one-level indirect addressing scheme. What sequence of page faults is incurred when all of the pages of a program are currently nonresident and the first instruction of the program is an indirect memory-load operation? What happens when the operating system is using a per-process frame allocation technique and only two pages are allocated to this process?
- 10.33** Consider the page references:



What pages represent the locality at time (X)?

- 10.34 Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?
- 10.35 A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then,

to replace a page, we can search for the page frame with the smallest counter.

- a. Define a page-replacement algorithm using this basic idea. Specifically address these problems:
 - What is the initial value of the counters?
 - When are counters increased?
 - When are counters decreased?
 - How is the page to be replaced selected?
 - b. How many page faults occur for your algorithm for the following reference string with four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.
 - c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?
- 10.36** Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory. Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?
- 10.37** What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
- 10.38** Is it possible for a process to have two working sets, one representing data and another representing code? Explain.
- 10.39** Consider the parameter Δ used to define the working-set window in the working-set model. When Δ is set to a low value, what is the effect on the page-fault frequency and the number of active (nonsuspended) processes currently executing in the system? What is the effect when Δ is set to a very high value?
- 10.40** In a 1,024-KB segment, memory is allocated using the buddy system. Using Figure 10.26 as a guide, draw a tree illustrating how the following memory requests are allocated:
- Request 5-KB
 - Request 135 KB.
 - Request 14 KB.
 - Request 3 KB.

- Request 12 KB.

Next, modify the tree for the following releases of memory. Perform coalescing whenever possible:

- Release 3 KB.
- Release 5 KB.
- Release 14 KB.
- Release 12 KB.

- 10.41** A system provides support for user-level and kernel-level threads. The mapping in this system is one to one (there is a corresponding kernel thread for each user thread). Does a multithreaded process consist of (a) a working set for the entire process or (b) a working set for each thread? Explain
- 10.42** The slab-allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn't scale well with multiple CPUs. What could be done to address this scalability issue?
- 10.43** Consider a system that allocates pages of different sizes to its processes. What are the advantages of such a paging scheme? What modifications to the virtual memory system would be needed to provide this functionality?

Programming Problems

- 10.44** Write a program that implements the FIFO, LRU, and optimal (OPT) page-replacement algorithms presented in Section 10.4. Have your program initially generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm, and record the number of page faults incurred by each algorithm. Pass the number of page frames to the program at startup. You may implement this program in any programming language of your choice. (You may find your implementation of either FIFO or LRU to be helpful in the virtual memory manager programming project.)

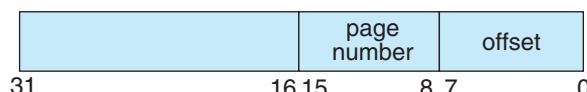
Programming Projects

Designing a Virtual Memory Manager

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65,536$ bytes. Your program will read from a file containing logical addresses and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. Your learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This will include resolving page faults using demand paging, managing a TLB, and implementing a page-replacement algorithm.

Specific

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) an 8-bit page offset. Hence, the addresses are structured as shown as:



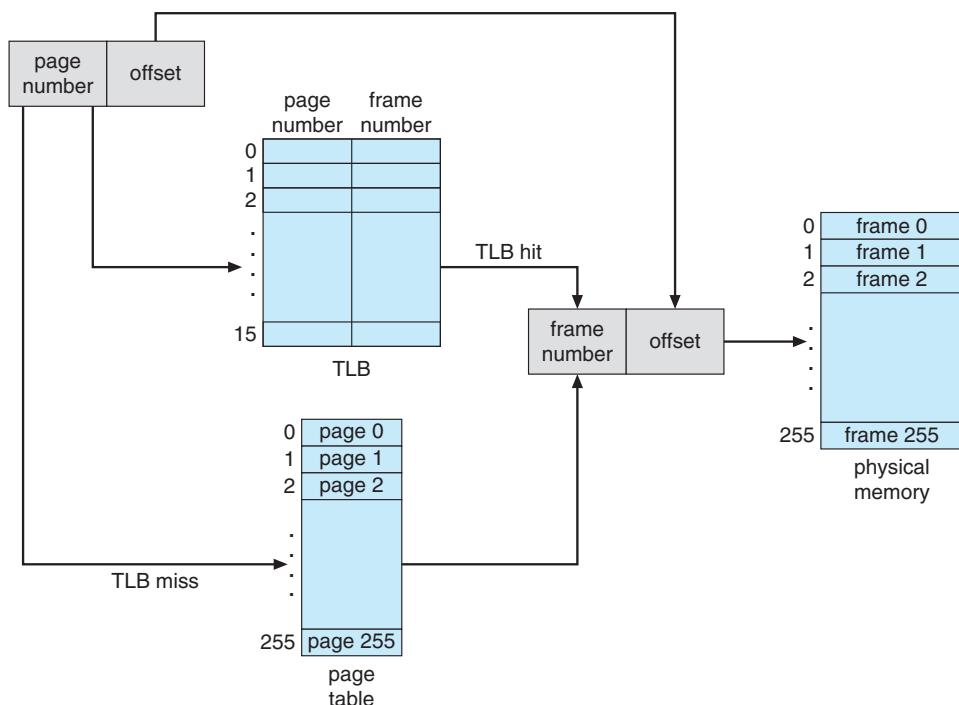
Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 65,536 bytes ($256 \text{ frames} \times 256\text{-byte frame size}$)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 9.3. First, the page number is extracted from the logical address, and the TLB is consulted. In the case of a TLB hit, the frame number is obtained from the TLB. In the case of a TLB miss, the page table must be consulted. In the latter case, either the frame number is obtained from the page table, or a page fault occurs. A visual representation of the address-translation process is:



Handling Page Faults

Your program will implement demand paging as described in Section 10.2. The backing store is represented by the file `BACKING_STORE.bin`, a binary file of size 65,536 bytes. When a page fault occurs, you will read in a 256-byte page from the file `BACKING_STORE` and store it in an available page frame in physical memory. For example, if a logical address with page number 15 resulted in a page fault, your program would read in page 15 from `BACKING_STORE` (remember that pages begin at 0 and are 256 bytes in size) and store it in a page frame in physical memory. Once this frame is stored (and the page table and TLB are updated), subsequent accesses to page 15 will be resolved by either the TLB or the page table.

You will need to treat BACKING_STORE.bin as a random-access file so that you can randomly seek to certain positions of the file for reading. We suggest using the standard C library functions for performing I/O, including `fopen()`, `fread()`, `fseek()`, and `fclose()`.

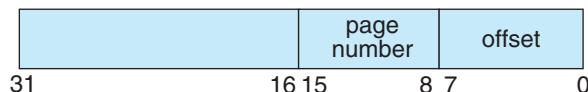
The size of physical memory is the same as the size of the virtual address space—65,536 bytes—so you do not need to be concerned about page replacements during a page fault. Later, we describe a modification to this project using a smaller amount of physical memory; at that point, a page-replacement strategy will be required.

Test File

We provide the file `addresses.txt`, which contains integer values representing logical addresses ranging from 0 to 65535 (the size of the virtual address space). Your program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.

How to Begin

First, write a simple program that extracts the page number and offset based on:



from the following integer numbers:

1, 256, 32768, 32769, 128, 65534, 33153

Perhaps the easiest way to do this is by using the operators for bit-masking and bit-shifting. Once you can correctly establish the page number and offset from an integer number, you are ready to begin.

Initially, we suggest that you bypass the TLB and use only a page table. You can integrate the TLB once your page table is working properly. Remember, address translation can work without a TLB; the TLB just makes it faster. When you are ready to implement the TLB, recall that it has only sixteen entries, so you will need to use a replacement strategy when you update a full TLB. You may use either a FIFO or an LRU policy for updating your TLB.

How to Run Your Program

Your program should run as follows:

```
./a.out addresses.txt
```

Your program will read in the file `addresses.txt`, which contains 1,000 logical addresses ranging from 0 to 65535. Your program is to translate each logical address to a physical address and determine the contents of the signed byte stored at the correct physical address. (Recall that in the C language, the `char` data type occupies a byte of storage, so we suggest using `char` values.)

Your program is to output the following values:

1. The logical address being translated (the integer value being read from `addresses.txt`).
2. The corresponding physical address (what your program translates the logical address to).
3. The signed byte value stored in physical memory at the translated physical address.

We also provide the file `correct.txt`, which contains the correct output values for the file `addresses.txt`. You should use this file to determine if your program is correctly translating logical to physical addresses.

Statistics

After completion, your program is to report the following statistics:

1. Page-fault rate—The percentage of address references that resulted in page faults.
2. TLB hit rate—The percentage of address references that were resolved in the TLB.

Since the logical addresses in `addresses.txt` were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

Page Replacement

Thus far, this project has assumed that physical memory is the same size as the virtual address space. In practice, physical memory is typically much smaller than a virtual address space. This phase of the project now assumes using a smaller physical address space with 128 page frames rather than 256. This change will require modifying your program so that it keeps track of free page frames as well as implementing a page-replacement policy using either FIFO or LRU (Section 10.4) to resolve page faults when there is no free memory.

File-System Interface



For most users, the file system is the most visible aspect of a general-purpose operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. Most file systems live on storage devices, which we described in Chapter 11 and will continue to discuss in the next chapter. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss the semantics of sharing files among multiple processes, users, and computers. Finally, we discuss ways to handle file protection, necessary when we have multiple users and want to control who may access files and how files may be accessed.

CHAPTER OBJECTIVES

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore file-system protection.

13.1 File Concept

Computers can store information on various storage media, such as NVM devices, HDDs, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

Because files are **the** method users and applications use to store and retrieve data, and because they are so general purpose, their use has stretched beyond its original confines. For example, UNIX, Linux, and some other operating systems provide a proc file system that uses file-system interfaces to provide access to system information (such as process details).

The information in a file is defined by its creator. Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type. A **text fil** is a sequence of characters organized into lines (and possibly pages). A **source fil** is a sequence of functions, each of which is further organized as declarations followed by executable statements. An **executable fil** is a series of code sections that the loader can bring into memory and execute.

13.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, and another user might edit that file by specifying its name. The file's owner might write the file to a USB drive, send it as an e-mail attachment, or copy it across a network, and it could still be called `example.c` on the destination system. Unless there is a sharing and synchronization method, that second copy is now independent of the first and can be changed separately.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name.** The symbolic file name is the only information kept in human-readable form.
- **Identifie**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type.** This information is needed for systems that support different types of files.
- **Location.** This information is a pointer to a device and to the location of the file on that device.

- **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- **Timestamps and user identification.** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

Some newer file systems also support **extended file attributes**, including character encoding of the file and security features such as a file checksum. Figure 13.1 illustrates a **file info window** on macOS that displays a file's attributes.

The information about all files is kept in the directory structure, which resides on the same device as the files themselves. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each file. In a system with many files, the size of the directory itself may be megabytes or gigabytes. Because directories must match the volatility of the files, like files, they must be stored on the device and are usually brought into memory piecemeal, as needed.

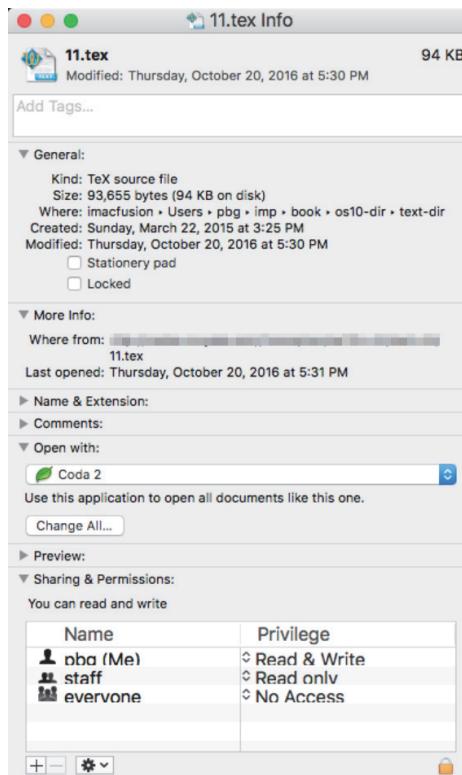


Figure 13.1 A file info window on macOS.

13.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these seven basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **Creating a fil** . Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 14. Second, an entry for the new file must be made in a directory.
- **Opening a fil** . Rather than have all file operations specify a file name, causing the operating system to evaluate the name, check access permissions, and so on, all operations except create and delete require a file `open()` first. If successful, the `open` call returns a file handle that is used as an argument in the other calls.
- **Writing a fil** . To write a file, we make a system call specifying both the open file handle and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place if it is sequential. The write pointer must be updated whenever a write occurs.
- **Reading a fil** . To read from a file, we use a system call that specifies the file handle and where (in memory) the next block of the file should be put. Again, the system needs to keep a **read pointer** to the location in the file where the next read is to take place, if sequential. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current-file-position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file** . The current-file-position pointer of the open file is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
- **Deleting a fil** . To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry. Note that some systems allow **hard links**—multiple names (directory entries) for the same file. In this case the actual file contents is not deleted until the last link is deleted.
- **Truncating a fil** . The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length. The file can then be reset to length zero, and its file space can be released.

These seven basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file. These primitive operations can then be combined to perform other file operations. For instance, we can create a copy of a file by creating a new file and then reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have operations that allow a user to determine the status of a file, such as the file's length, and to set file attributes, such as the file's owner.

As mentioned, most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an `open()` system call be made before a file is first used. The operating system keeps a table, called the **open-file table**, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table, potentially releasing locks. `create()` and `delete()` are system calls that work with closed rather than open files.

Some systems implicitly open a file when the first reference to it is made. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the `open()` system call before that file can be used. The `open()` operation takes a file name and searches the directory, copying the directory entry into the open-file table. The `open()` call can also accept access-mode information—create, read-only, read-write, append-only, and so on. This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process. The `open()` system call typically returns a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.

The implementation of the `open()` and `close()` operations is more complicated in an environment where several processes may open the file simultaneously. This may occur in a system where several different applications open the same file at the same time. Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an `open()` call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each `close()` decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

FILE LOCKING IN JAVA

In the Java API, acquiring a lock requires first obtaining the `FileChannel` for the file to be locked. The `lock()` method of the `FileChannel` is used to acquire the lock. The API of the `lock()` method is

```
FileLock lock(long begin, long end, boolean shared)
```

where `begin` and `end` are the beginning and ending positions of the region being locked. Setting `shared` to `true` is for shared locks; setting `shared` to `false` acquires the lock exclusively. The lock is released by invoking the `release()` of the `FileLock` returned by the `lock()` operation.

The program in Figure 13.2 illustrates file locking in Java. This program acquires two locks on the file `file.txt`. The lock for the first half of the file is an exclusive lock; the lock for the second half is a shared lock.

In summary, several pieces of information are associated with an open file.

- **File pointer.** On systems that do not include a file offset as part of the `read()` and `write()` system calls, the system must track the last read-write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count.** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Location of the file.** Most file operations require the system to read or write data within the file. The information needed to locate the file (wherever it is located, be it on mass storage, on a file server across the network, or on a RAM drive) is kept in memory so that the system does not have to read it from the directory structure for each operation.
- **Access rights.** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

Some operating systems provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes—for example, a system log file that can be accessed and modified by a number of processes in the system.

File locks provide functionality similar to reader-writer locks, covered in Section 7.1.2. A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock. It is important to note that not

```

import java.io.*;
import java.nio.channels.*;

public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;

    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;

        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

            // get the channel for the file
            FileChannel ch = raf.getChannel();

            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

            /** Now modify the data . . . */

            // release the lock
            exclusiveLock.release();

            // this locks the second half of the file - shared
            sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);

            /** Now read the data . . . */

            // release the lock
            sharedLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (exclusiveLock != null)
                exclusiveLock.release();
            if (sharedLock != null)
                sharedLock.release();
        }
    }
}

```

Figure 13.2 File-locking example in Java.

all operating systems provide both types of locks: some systems provide only exclusive file locking.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. With mandatory locking, once a process acquires an exclusive lock, the operating system will prevent any other process from

accessing the locked file. For example, assume a process acquires an exclusive lock on the file `system.log`. If we attempt to open `system.log` from another process—for example, a text editor—the operating system will prevent access until the exclusive lock is released. Alternatively, if the lock is advisory, then the operating system will not prevent the text editor from acquiring access to `system.log`. Rather, the text editor must be written so that it manually acquires the lock before accessing the file. In other words, if the locking scheme is mandatory, the operating system ensures locking integrity. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released. As a general rule, Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

The use of file locks requires the same precautions as ordinary process synchronization. For example, programmers developing on systems with mandatory locking must be careful to hold exclusive file locks only while they are accessing the file. Otherwise, they will prevent other processes from accessing the file as well. Furthermore, some measures must be taken to ensure that two or more processes do not become involved in a deadlock while trying to acquire file locks.

13.1.3 File Types

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to output the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period (Figure 13.3). In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include `resume.docx`, `server.c`, and `ReaderThread.cpp`.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a `.com`, `.exe`, or `.sh` extension can be executed, for instance. The `.com` and `.exe` files are two forms of binary executable files, whereas the `.sh` file is a **shell script** containing, in ASCII format, commands to the operating system. Application programs also use extensions to indicate file types in which they are interested. For example, Java compilers expect source files to have a `.java` extension, and the Microsoft Word word processor expects its files to end with a `.doc` or `.docx` extension. These extensions are not always required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered “hints” to the applications that operate on them.

Consider, too, the macOS operating system. In this system, each file has a type, such as `.app` (for application). Each file also has a creator attribute

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 13.3 Common file types.

containing the name of the program that created it. This attribute is set by the operating system during the `create()` call, so its use is enforced and supported by the system. For instance, a file produced by a word processor has the word processor's name as its creator. When the user opens that file, by double-clicking the mouse on the icon representing the file, the word processor is invoked automatically, and the file is loaded, ready to be edited.

The UNIX system uses a **magic number** stored at the beginning of some binary files to indicate the type of data in the file (for example, the format of an image file). Likewise, it uses a text magic number at the start of text files to indicate the type of file (which shell language a script is written in) and so on. (For more details on magic numbers and other computer jargon, see <http://www.catb.org/esr/jargon/>.) Not all files have magic numbers, so system features cannot be based solely on this information. UNIX does not record the name of the creating program, either. UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the operating system; they are meant mostly to aid users in determining what type of contents the file contains. Extensions can be used or ignored by a given application, but that is up to the application's programmer.

13.1.4 File Structure

File types also can be used to indicate the internal structure of the file. Source and object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that

is understood by the operating system. For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures.

This point brings us to one of the disadvantages of having the operating system support multiple file structures: it makes the operating system large and cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, it may be necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now, if we (as users) want to define an encrypted file to protect the contents from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines but rather is (apparently) random bits. Although it may appear to be a binary file, it is not executable. As a result, we may have to circumvent or misuse the operating system's file-type mechanism or abandon our encryption scheme.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, Windows, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure. However, all operating systems must support at least one structure—that of an executable file—so that the system is able to load and run programs.

13.1.5 Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

13.2 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Others (such as mainframe operating systems) support many access methods, and choosing the right one for a particular application is a major design problem.

13.2.1 Sequential Access

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

Reads and writes make up the bulk of the operations on a file. A read operation—`read_next()`—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—`write_next()`—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward n records for some integer n —perhaps only for $n = 1$. Sequential access, which is depicted in Figure 13.4, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

13.2.2 Direct Access

Another method is **direct access** (or **relative access**). Here, a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus,

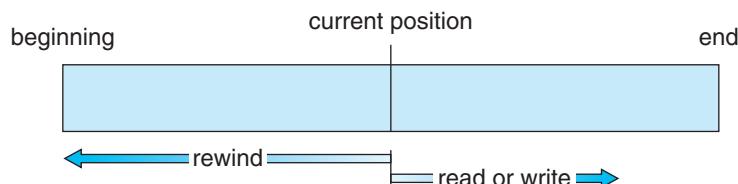


Figure 13.4 Sequential-access file.

we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names or search a small in-memory index to determine a block to read and search.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have `read(n)`, where n is the block number, rather than `read_next()`, and `write(n)` rather than `write_next()`. An alternative approach is to retain `read_next()` and `write_next()` and to add an operation `position_file(n)` where n is the block number. Then, to effect a `read(n)`, we would `position_file(n)` and then `read_next()`.

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the **allocation problem**, as we discuss in Chapter 14) and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

How, then, does the system satisfy a request for record N in a file? Assuming we have a logical record length L , the request for record N is turned into an I/O request for L bytes starting at location $L * (N)$ within the file (assuming the first record is $N = 0$). Since logical records are of a fixed size, it is also easy to read, write, or delete a record.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created. Such a file can be accessed only in a manner consistent with its declaration. We can easily simulate sequential access on a direct-access file by simply keeping a variable cp that defines our current position, as shown in Figure 13.5. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

13.2.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The **index**, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

sequential access	implementation for direct access
reset	<code>cp = 0;</code>
read_next	<code>read cp;</code> <code>cp = cp + 1;</code>
write_next	<code>write cp;</code> <code>cp = cp + 1;</code>

Figure 13.5 Simulation of sequential access on a direct-access file.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record. If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can make a binary search of the index. From this search, we learn exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/O.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 13.6 shows a similar situation as implemented by OpenVMS index and relative files.

13.3 Directory Structure

The directory can be viewed as a symbol table that translates file names into their file control blocks. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar

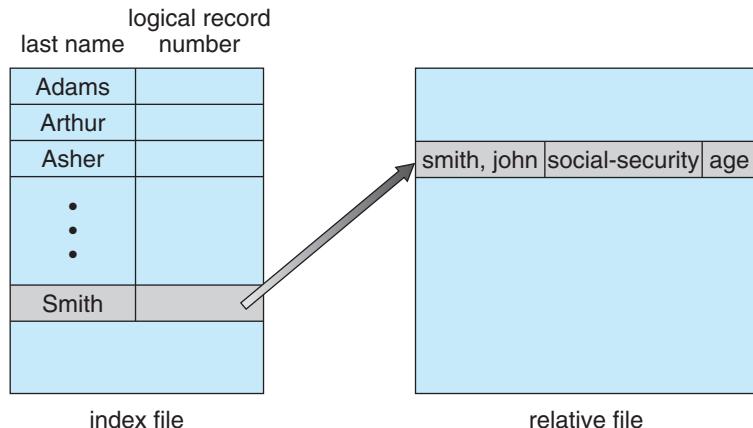


Figure 13.6 Example of index and relative files.

names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

- **Create a file.** New files need to be created and added to the directory.
- **Delete a file.** When a file is no longer needed, we want to be able to remove it from the directory. Note a delete leaves a hole in the directory structure and the file system may have a method to defragment the directory structure.
- **List a directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file.** Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system.** We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape, other secondary storage, or across a network to another system or the cloud. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to the backup target and the disk space of that file released for reuse by another file.

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

13.3.1 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 13.7).

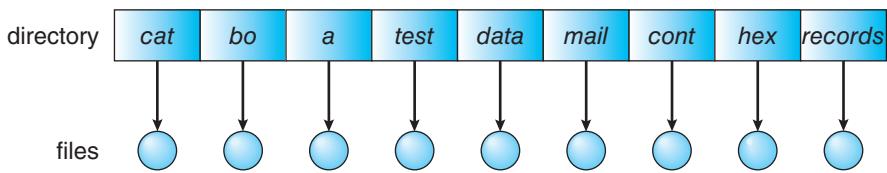


Figure 13.7 Single-level directory.

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file `test.txt`, then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment `prog2.c`; another 11 called it `assign2.c`. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

13.3.2 Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user fil directory (UFD)**. The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master fil directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 13.8).

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name

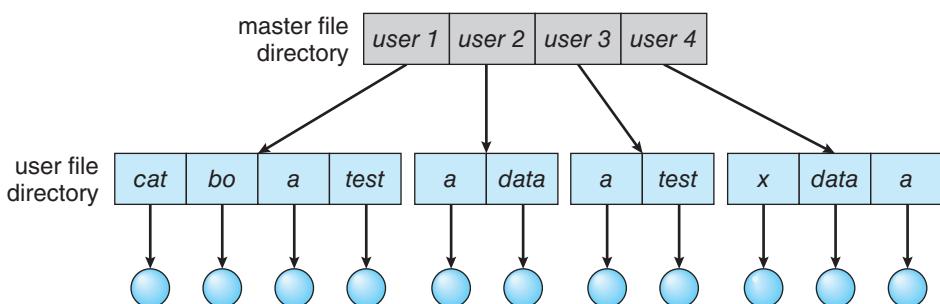


Figure 13.8 Two-level directory structure.

exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques discussed in Chapter 14 for files themselves.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a **path name**. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

For example, if user A wishes to access her own test file named `test.txt`, she can simply refer to `test.txt`. To access the file named `test.txt` of user B (with directory-entry name `userb`), however, she might have to refer to `/userb/test.txt`. Every system has its own syntax for naming files in directories other than the user's own.

Additional syntax is needed to specify the volume of a file. For instance, in Windows a volume is specified by a letter followed by a colon. Thus, a file specification might be `C:\userb\test`. Some systems go even further and separate the volume, directory name, and file name parts of the specification. In OpenVMS, for instance, the file `login.com` might be specified as: `u:[sst.crissmeyer]login.com;1`, where `u` is the name of the volume, `sst` is the name of the directory, `crissmeyer` is the name of the subdirectory, and `1` is the version number. Other systems—such as UNIX and Linux—simply treat the volume name as part of the directory name. The first name given is that of the volume, and the rest is the directory and file. For instance, `/u/pgalvin/test` might specify volume `u`, directory `pgalvin`, and file `test`.

A special instance of this situation occurs with the system files. Programs provided as part of the system—loaders, assemblers, compilers, utility routines, libraries, and so on—are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and executed. Many command interpreters simply treat such a command as the name of a file to load and execute. In the directory system as we defined it above, this file name would be searched for in the current UFD. One solution would be to copy the system files into each UFD. However, copying all the system files would waste an enormous amount of space. (If the system files

require 5 MB, then supporting 12 users would require $5 \times 12 = 60$ MB just for copies of the system files.)

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the **search path**. The search path can be extended to contain an unlimited list of directories to search when a command name is given. This method is the one most used in UNIX and Windows. Systems can also be designed so that each user has his own search path.

13.3.3 Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 13.9). This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. In many implementations, a directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special

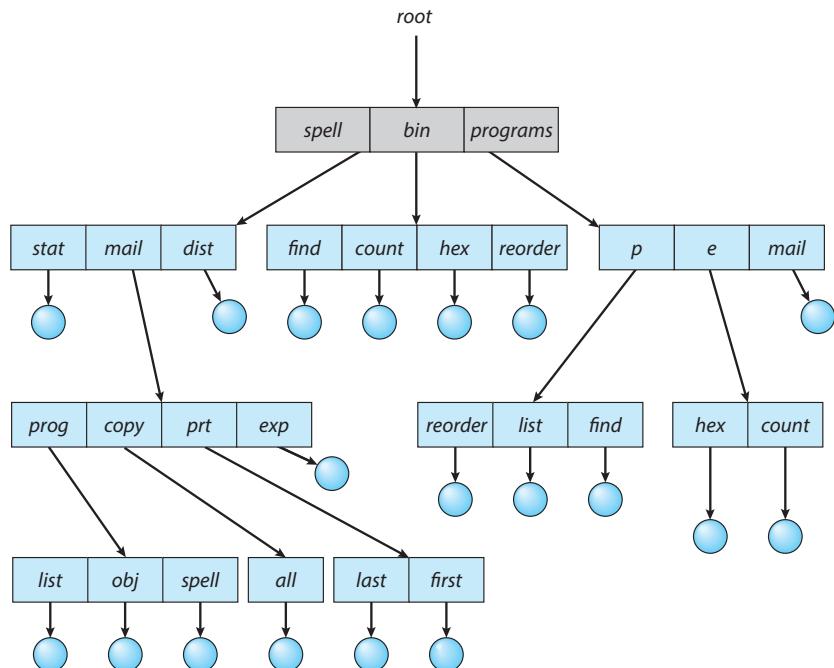


Figure 13.9 Tree-structured directory structure.

system calls are used to create and delete directories. In this case the operating system (or the file system code) implements another file format, that of a directory.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call could be provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants. Other systems leave it to the application (say, a shell) to track and operate on a current directory, as each process could have different current directories.

The initial current directory of a user's login shell is designated when the user job starts or the user logs in. The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user that specifies the user's initial current directory. From that shell, other processes can be spawned. The current directory of any subprocess is usually the current directory of the parent when it was spawned.

Path names can be of two types: absolute and relative. In UNIX and Linux, an **absolute path name** begins at the root (which is designated by an initial "/") and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure 13.9, if the current directory is /spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name /spell/mail/prt/first.

Allowing a user to define her own subdirectories permits her to impose a structure on her files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information. For example, the directory **programs** may contain source programs; the directory **bin** may store all the binaries. (As a side note, executable files were known in many systems as "binaries" which led to them being stored in the **bin** directory.)

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX **rm** command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command. If that command

is issued in error, a large number of files and directories will need to be restored (assuming a backup exists).

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path name. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user A's directory and access the file by its file name.

13.3.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be *shared*. A shared directory or file exists in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An **acyclic graph**—that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 13.10). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is

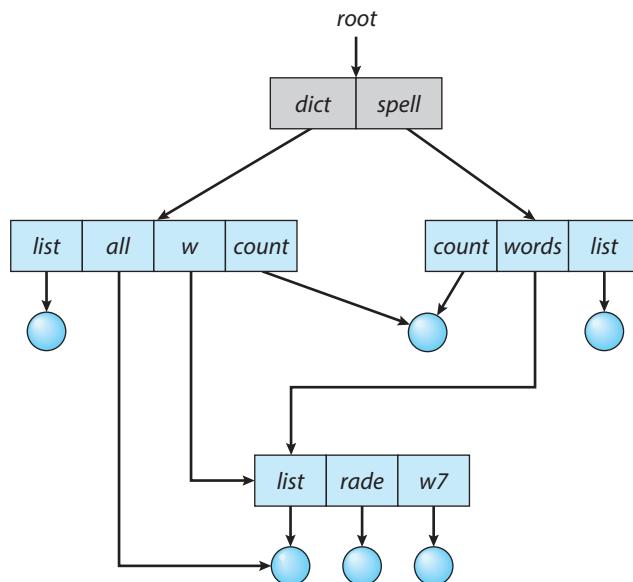


Figure 13.10 Acyclic-graph directory structure.

particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share can be put into one directory. The home directory of each team member could contain this directory of shared files as a subdirectory. Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by UNIX systems, is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information. We **resolve** the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry (or by having a special type on systems that support types) and are effectively indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. Consider the difference between this approach and the creation of a link. The link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency when a file is modified.

An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system—to find a file, to accumulate statistics on all files, or to copy all files to backup storage—this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist and can fail to resolve the link name; the access is treated just as with any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is

deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced. Microsoft Windows uses the same approach.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list—we need to keep only a count of the number of references. Adding a new link or directory entry increments the reference count. Deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links (or [hard links](#)), keeping a reference count in the file information block (or inode; see Section C.7.2). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid problems such as the ones just discussed, some systems simply do not allow shared directories or links.

13.3.5 General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure (Figure 13.11).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure. In this case, we generally need to use a [garbage collection](#)

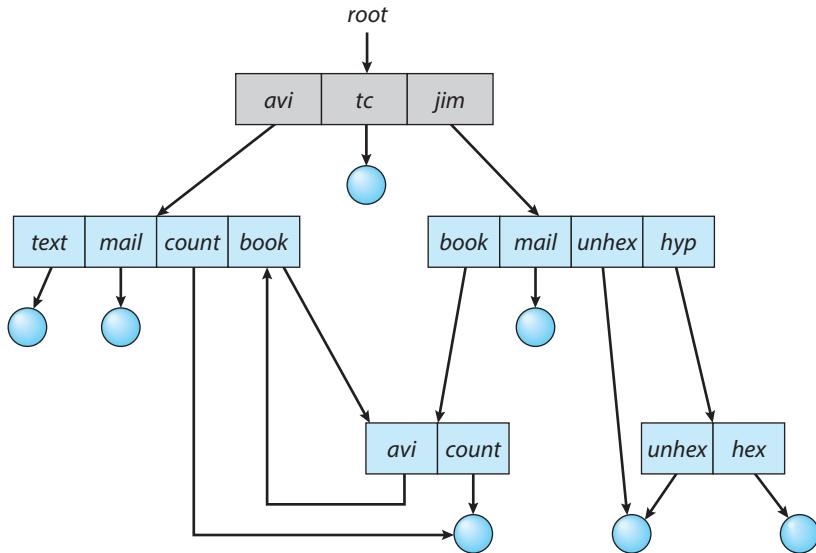


Figure 13.11 General graph directory.

scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. A simpler algorithm in the special case of directories and links is to bypass links during directory traversal. Cycles are avoided, and no extra overhead is incurred.

13.4 Protection

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes,

and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Reliability was covered in more detail in Chapter 11.

Protection can be provided in many ways. For a laptop system running a modern operating system, we might provide protection by requiring a user name and password authentication to access it, encrypting the secondary storage so even someone opening the laptop and removing the drive would have a difficult time accessing its data, and firewalling network access so that when it is in use it is difficult to break in via its network connection. In multiuser systems, even valid access of the system needs more advanced mechanisms to allow only valid access of the data.

13.4.1 Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read.** Read from the file.
- **Write.** Write or rewrite the file.
- **Execute.** Load the file into memory and execute it.
- **Append.** Write new information at the end of the file.
- **Delete.** Delete the file and free its space for possible reuse.
- **List.** List the name and attributes of the file.
- **Attribute change.** Changing the attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

Many protection mechanisms have been proposed. Each has advantages and disadvantages and must be appropriate for its intended application. A small computer system that is used by only a few members of a research group, for example, may not need the same types of protection as a large corporate computer that is used for research, finance, and personnel operations. We discuss some approaches to protection in the following sections and present a more complete treatment in Chapter 17.

13.4.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner.** The user who created the file is the owner.
- **Group.** A set of users who are sharing the file and need similar access is a group, or work group.
- **Other.** All other users in the system.

The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access-control scheme just described. For example, Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named `book.tex`. The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)

PERMISSIONS IN A UNIX SYSTEM

In the UNIX system, directory protection and file protection are handled similarly. Associated with each file and directory are three fields—owner, group, and universe—each consisting of the three bits **rwx**, where **r** controls read access, **w** controls write access, and **x** controls execution. Thus, a user can list the content of a subdirectory only if the **r** bit is set in the appropriate field. Similarly, a user can change his current directory to another current directory (say, **foo**) only if the **x** bit associated with the **foo** subdirectory is set in the appropriate field.

A sample directory listing from a UNIX environment is shown in below:

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	jwg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2017	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2017	program
drwx--x--x	4	tag	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

The first field describes the protection of the file or directory. A **d** as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in bytes, the date of last modification, and finally the file's name (with optional extension).

To achieve such protection, we must create a new group—say, **text**—with members Jim, Dawn, and Jill. The name of the group, **text**, must then be associated with the file **book.tex**, and the access rights must be set in accordance with the policy we have outlined.

Now consider a visitor to whom Sara would like to grant temporary access to Chapter 1. The visitor cannot be added to the **text** group because that would give him access to all chapters. Because a file can be in only one group, Sara cannot add another group to Chapter 1. With the addition of access-control-list functionality, though, the visitor can be added to the access control list of Chapter 1.

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways. For example, in the UNIX system, groups can be created and modified only by the manager of the facility (or by any superuser). Thus, control is achieved through human interaction. Access lists are discussed further in Section 17.6.2.

With the more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it. For example, the UNIX system defines three fields of three bits each—**rwx**, where **r** controls read access, **w** controls write access, and **x** controls execution. A separate field is kept for the

file owner, for the file’s group, and for all other users. In this scheme, nine bits per file are needed to record protection information. Thus, for our example, the protection fields for the file `book.tex` are as follows: for the owner `Sara`, all bits are set; for the group `text`, the `r` and `w` bits are set; and for the universe, only the `r` bit is set.

One difficulty in combining approaches comes in the user interface. Users must be able to tell when the optional ACL permissions are set on a file. In the Solaris example, a “+” is appended to the regular permissions, as in:

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

A separate set of commands, `setfacl` and `getfacl`, is used to manage the ACLs.

Windows users typically manage access-control lists via the GUI. Figure 13.12 shows a file-permission window on Windows 7 NTFS file system. In this example, user “guest” is specifically denied access to the file `ListPanel.java`.

Another difficulty is assigning precedence when permission and ACLs conflict. For example, if Walter is in a file’s group, which has read permission, but the file has an ACL granting Walter read and write permission, should a write by Walter be granted or denied? Solaris and other operating systems give ACLs precedence (as they are more fine-grained and are not assigned by default). This follows the general rule that specificity should have priority.

13.4.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem. More commonly encryption of a partition or individual files provides strong protection, but password management is key.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file is significant in itself. Thus, listing the contents of a directory must be a protected operation. Similarly, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic and general graphs), a given user may have different access rights to a particular file, depending on the path name used.

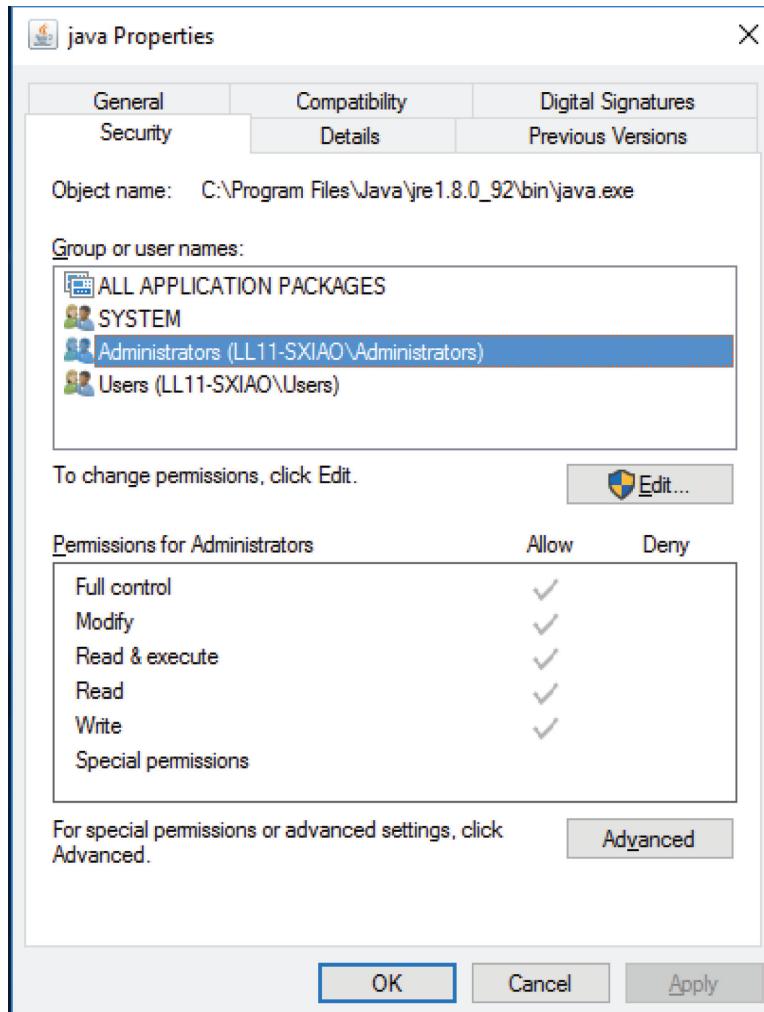


Figure 13.12 Windows 10 access-control list management.

13.5 Memory-Mapped Files

There is one other method of accessing files, and it is very commonly used. Consider a sequential read of a file on disk using the standard system calls `open()`, `read()`, and `write()`. Each file access requires a system call and disk access. Alternatively, we can use the virtual memory techniques discussed in Chapter 10 to treat file I/O as routine memory accesses. This approach, known as **memory mapping** a file, allows a part of the virtual address space to be logically associated with the file. As we shall see, this can lead to significant performance increases.

13.5.1 Basic Mechanism

Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds through ordinary demand

paging, resulting in a page fault. However, a page-sized portion of the file is read from the file system into a physical page (some systems may opt to read in more than a page-sized chunk of memory at a time). Subsequent reads and writes to the file are handled as routine memory accesses. Manipulating files through memory rather than incurring the overhead of using the `read()` and `write()` system calls simplifies and speeds up file access and usage.

Note that writes to the file mapped in memory are not necessarily immediate (synchronous) writes to the file on secondary storage. Generally, systems update the file based on changes to the memory image only when the file is closed. Under memory pressure, systems will have any intermediate changes to swap space to not lose them when freeing memory for other uses. When the file is closed, all the memory-mapped data are written back to the file on secondary storage and removed from the virtual memory of the process.

Some operating systems provide memory mapping only through a specific system call and use the standard system calls to perform all other file I/O. However, some systems choose to memory-map a file regardless of whether the file was specified as memory-mapped. Let's take Solaris as an example. If a file is specified as memory-mapped (using the `mmap()` system call), Solaris maps the file into the address space of the process. If a file is opened and accessed using ordinary system calls, such as `open()`, `read()`, and `write()`, Solaris still memory-maps the file; however, the file is mapped to the kernel address space. Regardless of how the file is opened, then, Solaris treats all file I/O as memory-mapped, allowing file access to take place via the efficient memory subsystem and avoiding system call overhead caused by each traditional `read()` and `write()`.

Multiple processes may be allowed to map the same file concurrently, to allow sharing of data. Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file. Given our earlier discussions of virtual memory, it should be clear how the sharing of memory-mapped sections of memory is implemented: the virtual memory map of each sharing process points to the same page of physical memory—the page that holds a copy of the disk block. This memory sharing is illustrated in Figure 13.13. The memory-mapping system calls can also support copy-on-write functionality, allowing processes to share a file in read-only mode but to have their own copies of any data they modify. So that access to the shared data is coordinated, the processes involved might use one of the mechanisms for achieving mutual exclusion described in Chapter 6.

Quite often, shared memory is in fact implemented by memory mapping files. Under this scenario, processes can communicate using shared memory by having the communicating processes memory-map the same file into their virtual address spaces. The memory-mapped file serves as the region of shared memory between the communicating processes (Figure 13.14). We have already seen this in Section 3.5, where a POSIX shared-memory object is created and each communicating process memory-maps the object into its address space. In the following section, we discuss support in the Windows API for shared memory using memory-mapped files.

13.5.2 Shared Memory in the Windows API

The general outline for creating a region of shared memory using memory-mapped files in the Windows API involves first creating a [file mapping](#) for the

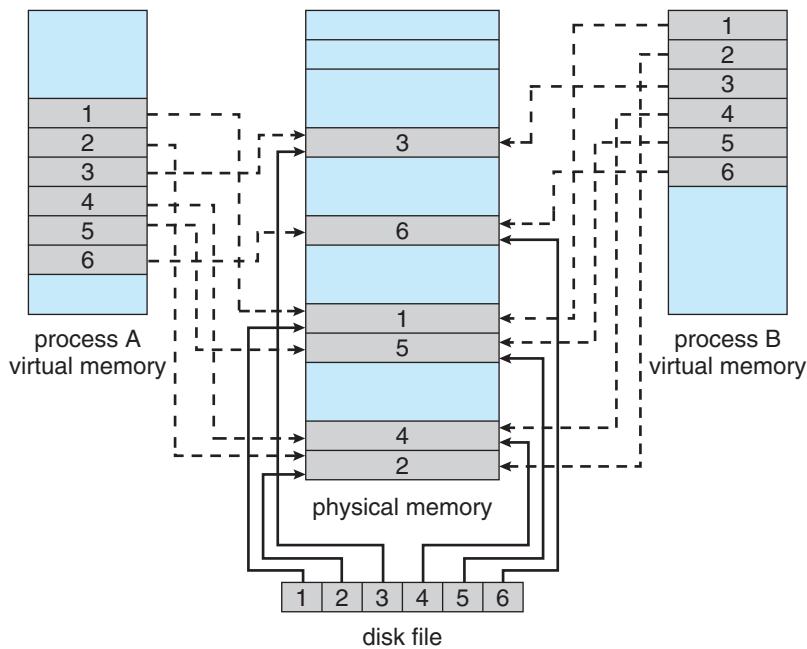


Figure 13.13 Memory-mapped files.

file to be mapped and then establishing a [view](#) of the mapped file in a process's virtual address space. A second process can then open and create a view of the mapped file in its virtual address space. The mapped file represents the shared-memory object that will enable communication to take place between the processes.

We next illustrate these steps in more detail. In this example, a producer process first creates a shared-memory object using the memory-mapping features available in the Windows API. The producer then writes a message to shared memory. After that, a consumer process opens a mapping to the shared-memory object and reads the message written by the consumer.

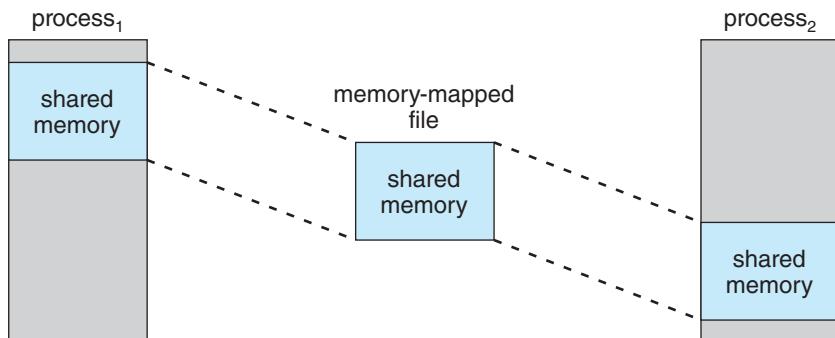


Figure 13.14 Shared memory using memory-mapped I/O.

To establish a memory-mapped file, a process first opens the file to be mapped with the `CreateFile()` function, which returns a HANDLE to the opened file. The process then creates a mapping of this file HANDLE using the `CreateFileMapping()` function. Once the file mapping is done, the process establishes a view of the mapped file in its virtual address space with the `MapViewOfFile()` function. The view of the mapped file represents the portion of the file being mapped in the virtual address space of the process—the entire file or only a portion of it may be mapped. This sequence in the program

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hFile, hMapFile;
    LPVOID lpMapAddress;

    hFile = CreateFile("temp.txt", /* file name */
                      GENERIC_READ | GENERIC_WRITE, /* read/write access */
                      0, /* no sharing of the file */
                      NULL, /* default security */
                      OPEN_ALWAYS, /* open new or existing file */
                      FILE_ATTRIBUTE_NORMAL, /* routine file attributes */
                      NULL); /* no file template */

    hMapFile = CreateFileMapping(hFile, /* file handle */
                                NULL, /* default security */
                                PAGE_READWRITE, /* read/write access to mapped pages */
                                0, /* map entire file */
                                0,
                                TEXT("SharedObject")); /* named shared memory object */

    lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
                               FILE_MAP_ALL_ACCESS, /* read/write access */
                               0, /* mapped view of entire file */
                               0,
                               0);

    /* write to shared memory */
    sprintf(lpMapAddress, "Shared memory message");

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hFile);
    CloseHandle(hMapFile);
}
```

Figure 13.15 Producer writing to shared memory using the Windows API.

is shown in Figure 13.15. (We eliminate much of the error checking for code brevity.)

The call to `CreateFileMapping()` creates a **named shared-memory object** called `SharedObject`. The consumer process will communicate using this shared-memory segment by creating a mapping to the same named object. The producer then creates a view of the memory-mapped file in its virtual address space. By passing the last three parameters the value 0, it indicates that the mapped view is the entire file. It could instead have passed values specifying an offset and size, thus creating a view containing only a subsection of the file. (It is important to note that the entire mapping may not be loaded into memory when the mapping is established. Rather, the mapped file may be demand-paged, thus bringing pages into memory only as they are accessed.) The `MapViewOfFile()` function returns a pointer to the shared-memory object; any accesses to this memory location are thus accesses to the memory-mapped file. In this instance, the producer process writes the message “Shared memory message” to shared memory.

A program illustrating how the consumer process establishes a view of the named shared-memory object is shown in Figure 13.16. This program is

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hMapFile;
    LPVOID lpMapAddress;

    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W access */
        FALSE, /* no inheritance */
        TEXT("SharedObject")); /* name of mapped file object */

    lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
        FILE_MAP_ALL_ACCESS, /* read/write access */
        0, /* mapped view of entire file */
        0,
        0);

    /* read from shared memory */
    printf("Read message %s", lpMapAddress);

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hMapFile);
}
```

Figure 13.16 Consumer reading from shared memory using the Windows API.

somewhat simpler than the one shown in Figure 13.15, as all that is necessary is for the process to create a mapping to the existing named shared-memory object. The consumer process must also create a view of the mapped file, just as the producer process did in the program in Figure 13.15. The consumer then reads from shared memory the message “Shared memory message” that was written by the producer process.

Finally, both processes remove the view of the mapped file with a call to `UnmapViewOfFile()`. We provide a programming exercise at the end of this chapter using shared memory with memory mapping in the Windows API.

13.6 Summary

- A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (of fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.
- A major task for the operating system is to map the logical file concept onto physical storage devices such as hard disk or NVM device. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.
- Within a file system, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user’s files. The directory lists the files by name and includes the file’s location on the disk, length, type, owner, time of creation, time of last use, and so on.
- The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize files. Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.
- Remote file systems present challenges in reliability, performance, and security. Distributed information systems maintain user, host, and access information so that clients and servers can share state information to manage use and access.
- Since files are the main information-storage mechanism in most computer systems, file protection is needed on multiuser systems. Access to files can be controlled separately for each type of access—read, write, execute, append, delete, list directory, and so on. File protection can be provided by access lists, passwords, or other techniques.

Practice Exercises

- 13.1 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept. Other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.
- 13.2 Why do some systems keep track of the type of a file, while still others leave it to the user and others simply do not implement multiple file types? Which system is “better”?
- 13.3 Similarly, some systems support many types of structures for a file’s data, while others simply support a stream of bytes. What are the advantages and disadvantages of each approach?
- 13.4 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation’s success. How would your answer change if file names were limited to seven characters?
- 13.5 Explain the purpose of the `open()` and `close()` operations.
- 13.6 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.
 - a. Describe the protection problems that could arise.
 - b. Suggest a scheme for dealing with each of these protection problems.
- 13.7 Consider a system that supports 5,000 users. Suppose that you want to allow 4,990 of these users to be able to access one file.
 - a. How would you specify this protection scheme in UNIX?
 - b. Can you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?
- 13.8 Researchers have suggested that, instead of having an access-control list associated with each file (specifying which users can access the file, and how), we should have a **user control list** associated with each user (specifying which files a user can access, and how). Discuss the relative merits of these two schemes.

Further Reading

A multilevel directory structure was first implemented on the MULTICS system ([Organick (1972)]). Most operating systems now implement multilevel directory structures. These include Linux ([Love (2010)]), macOS ([Singh (2007)]), Solaris ([McDougall and Mauro (2007)]), and all versions of Windows ([Russinovich et al. (2017)]).

A general discussion of Solaris file systems is found in the *Sun System Administration Guide: Devices and File Systems* (<http://docs.sun.com/app/docs/doc/817-5093>).

The network file system (NFS), designed by Sun Microsystems, allows directory structures to be spread across networked computer systems. NFS Version 4 is described in RFC3505 (<http://www.ietf.org/rfc/rfc3530.txt>).

A great source of the meanings of computer jargon is <http://www.catb.org/esr/jargon/>.

Bibliography

- [Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).
- [McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).
- [Organick (1972)] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).
- [Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).
- [Singh (2007)] A. Singh, *Mac OS X Internals: A Systems Approach*, Addison-Wesley (2007).

Chapter 13 Exercises

- 13.9 Consider a file system in which a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?
- 13.10 The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate table for each user or maintain just one table that contains references to files that are currently being accessed by all users? If the same file is being accessed by two different programs or users, should there be separate entries in the open-file table? Explain.
- 13.11 What are the advantages and disadvantages of providing mandatory locks instead of advisory locks whose use is left to users' discretion?
- 13.12 Provide examples of applications that typically access files according to the following methods:
 - Sequential
 - Random
- 13.13 Some systems automatically open a file when it is referenced for the first time and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme compared with the more traditional one, where the user has to open and close the file explicitly.
- 13.14 If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?
- 13.15 Give an example of an application that could benefit from operating-system support for random access to indexed files.
- 13.16 Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

File-System Implementation



As we saw in Chapter 13, the file system provides the mechanism for on-line storage and access to file contents, including data and programs. File systems usually reside permanently on secondary storage, which is designed to hold a large amount of data. This chapter is primarily concerned with issues surrounding file storage and access on the most common secondary-storage media, hard disk drives and nonvolatile memory devices. We explore ways to structure file use, to allocate storage space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage. Performance issues are considered throughout the chapter.

A given general-purpose operating system provides several file systems. Additionally, many operating systems allow administrators or users to add file systems. Why so many? File systems vary in many respects, including features, performance, reliability, and design goals, and different file systems may serve different purposes. For example, a temporary file system is used for fast storage and retrieval of nonpersistent files, while the default secondary storage file system (such as Linux ext4) sacrifices performance for reliability and features. As we've seen throughout this study of operating systems, there are plenty of choices and variations, making thorough coverage a challenge. In this chapter, we concentrate on the common denominators.

CHAPTER OBJECTIVES

- Describe the details of implementing local file systems and directory structures.
- Discuss block allocation and free-block algorithms and trade-offs.
- Explore file system efficiency and performance issues.
- Look at recovery from file system failures.
- Describe the WAFL file system as a concrete example.

14.1 File-System Structure

Disk provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same block.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires the drive moving the read–write heads and waiting for the media to rotate.

Nonvolatile memory (NVM) devices are increasingly used for file storage and thus as a location for file systems. They differ from hard disks in that they cannot be rewritten in place and they have different performance characteristics. We discuss disk and NVM-device structure in detail in Chapter 11.

To improve I/O efficiency, I/O transfers between memory and mass storage are performed in units of **blocks**. Each block on a hard disk drive has one or more sectors. Depending on the disk drive, sector size is usually 512 bytes or 4,096 bytes. NVM devices usually have blocks of 4,096 bytes, and the transfer methods used are similar to those used by disk drives.

File systems provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure 14.1 is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands, such as “retrieve block 123.” Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller which device location to act on and what actions to take. The details of device drivers and the I/O infrastructure are covered in Chapter 12.

The **basic file system** (called the “block I/O subsystem” in Linux) needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device. It issues commands to the drive based on logical block addresses. It is also concerned with I/O request scheduling. This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks. A block in the buffer is allocated before the transfer of a mass storage block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a

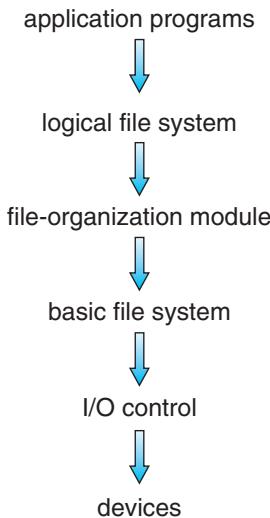


Figure 14.1 Layered file system.

requested I/O to complete. Caches are used to hold frequently used file-system metadata to improve performance, so managing their contents is critical for optimum system performance.

The **file-organization module** knows about files and their logical blocks. Each file's logical blocks are numbered from 0 (or 1) through N . The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks. A **file control block (FCB)** (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection, as discussed in Chapters 13 and 17.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file-system and file-organization modules. Unfortunately, layering can introduce more operating-system overhead, which may result in decreased performance. The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

Many file systems are in use today, and most operating systems support more than one. For example, most CD-ROMs are written in the ISO 9660 format, a standard format agreed on by CD-ROM manufacturers. In addition to removable-media file systems, each operating system has one or more disk-based file systems. UNIX uses the **UNIX file system (UFS)**, which is based on the

Berkeley Fast File System (FFS). Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats. Although Linux supports over 130 different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4. There are also distributed file systems in which a file system on a server is mounted by one or more client computers across a network.

File-system research continues to be an active area of operating-system design and implementation. Google created its own file system to meet the company's specific storage and retrieval needs, which include high-performance access from many clients across a very large number of disks. Another interesting project is the FUSE file system, which provides flexibility in file-system development and use by implementing and executing file systems as user-level rather than kernel-level code. Using FUSE, a user can add a new file system to a variety of operating systems and can use that file system to manage her files.

14.2 File-System Operations

As was described in Section 13.1.2, operating systems implement `open()` and `close()` systems calls for processes to request access to file contents. In this section, we delve into the structures and operations used to implement file-system operations.

14.2.1 Overview

Several on-storage and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system, but some general principles apply.

On storage, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files. Many of these structures are detailed throughout the remainder of this chapter. Here, we describe them briefly:

- A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the **boot block**. In NTFS, it is the **partition boot sector**.
- A **volume control block** (per volume) contains volume details, such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a **superblock**. In NTFS, it is stored in the **master file table**.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.

- A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.

- An in-memory **mount table** contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
- The **system-wide open-fil table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-fil table** contains pointers to the appropriate entries in the system-wide open-file table, as well as other information, for all files the process has open.
- Buffers hold file-system blocks when they are being read from or written to a file system.

To create a new file, a process calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.) The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the file system. A typical FCB is shown in Figure 14.2.

Some operating systems, including UNIX, treat a directory exactly the same as a file—one with a “type” field indicating that it is a directory. Other oper-

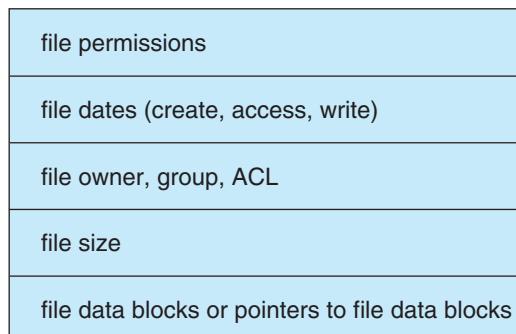


Figure 14.2 A typical file-control block.

ating systems, including Windows, implement separate system calls for files and directories and treat directories as entities separate from files. Whatever the larger structural issues, the logical file system can call the file-organization module to map the directory I/O into storage block locations, which are passed on to the basic file system and I/O control system.

14.2.2 Usage

Now that a file has been created, it can be used for I/O. First, though, it must be opened. The `open()` call passes a file name to the logical file system. The `open()` system call first searches the system-wide open-file table to see if the file is already in use by another process. If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table. This algorithm can save substantial overhead. If the file is not already open, the directory structure is searched for the given file name. Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.

Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields may include a pointer to the current location in the file (for the next `read()` or `write()` operation) and the access mode in which the file is open. The `open()` call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer. The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk. It could be cached, though, to save time on subsequent opens of the same file. The name given to the entry varies. UNIX systems refer to it as a **fil descriptor**; Windows refers to it as a **fil handle**.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented. When all users that have opened the file close it, any updated metadata are copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.

The caching aspects of file-system structures should not be overlooked. Most systems keep all information about an open file, except for its actual data blocks, in memory. The BSD UNIX system is typical in its use of caches wherever disk I/O can be saved. Its average cache hit rate of 85 percent shows that these techniques are well worth implementing. The BSD UNIX system is described fully in Appendix C.

The operating structures of a file-system implementation are summarized in Figure 14.3.

14.3 Directory Implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. In this section, we discuss the trade-offs involved in choosing one of these algorithms.

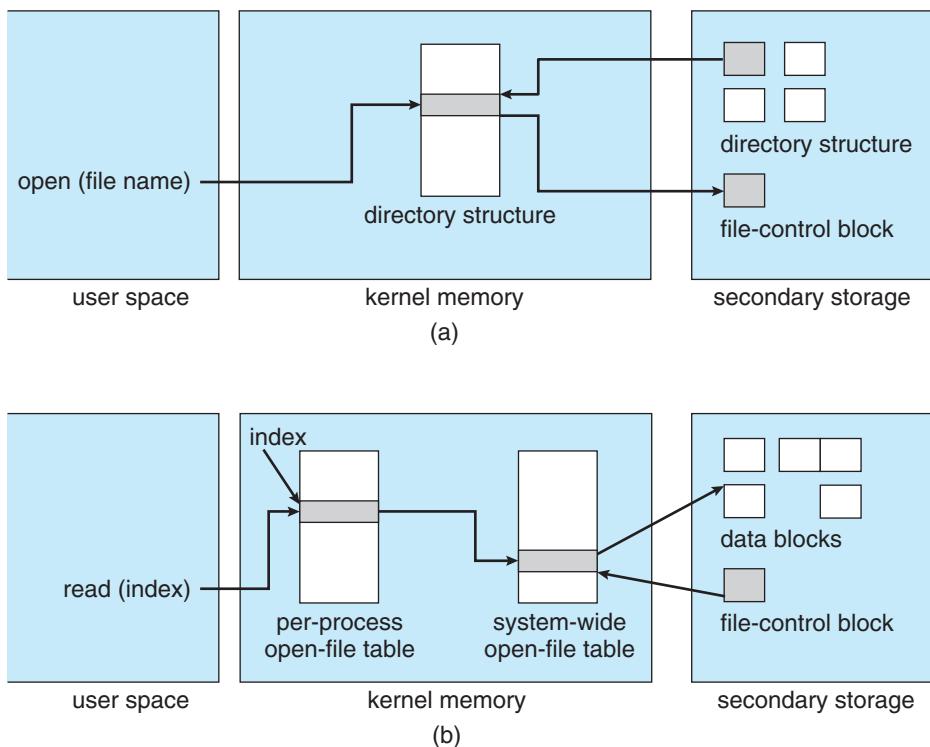


Figure 14.3 In-memory file-system structures. (a) File open. (b) File read.

14.3.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file and then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, assigning it an invalid inode number (such as 0), or by including a used–unused bit in each entry), or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time required to delete a file.

The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly reread the information from secondary storage. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of

directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a balanced tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

14.3.2 Hash Table

Another data structure used for a file directory is a hash table. Here, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63 (for instance, by using the remainder of a division by 64). If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

Alternatively, we can use a chained-overflow hash table. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries. Still, this method is likely to be much faster than a linear search through the entire directory.

14.4 Allocation Methods

The direct-access nature of secondary storage gives us flexibility in the implementation of files. In almost every case, many files are stored on the same device. The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly. Three major methods of allocating secondary storage space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

14.4.1 Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the device. Device addresses define a linear ordering on the device. With this ordering, assuming that only one job is accessing the device, accessing block $b + 1$ after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, for HDDs, the number of disk seeks required for accessing contiguously allocated files is

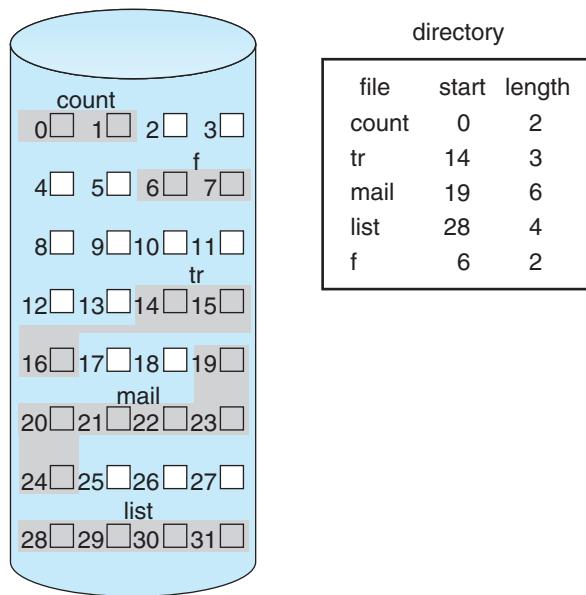


Figure 14.4 Contiguous allocation of disk space.

minimal (assuming blocks with close logical addresses are close physically), as is seek time when a seek is finally needed.

Contiguous allocation of a file is defined by the address of the first block and length (in block units) of the file. If the file is n blocks long and starts at location b , then it occupies blocks $b, b + 1, b + 2, \dots, b + n - 1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure 14.4). Contiguous allocation is easy to implement but has limitations, and is therefore not used in modern file systems.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b , we can immediately access block $b + i$. Thus, both sequential and direct access can be supported by contiguous allocation.

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished; these management systems are discussed in Section 14.5. Any management system can be used, but some are slower than others.

The contiguous-allocation problem can be seen as a particular application of the general **dynamic storage-allocation** problem discussed in Section 9.2, which involves how to satisfy a request of size n from a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

All these algorithms suffer from the problem of **external fragmentation**. As files are allocated and deleted, the free storage space is broken into little pieces.

External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

One strategy for preventing loss of significant amounts of storage space to external fragmentation is to copy an entire file system onto another device. The original device is then freed completely, creating one large contiguous free space. We then copy the files back onto the original device by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time, however, and the cost can be particularly high for large storage devices. Compacting these devices may take hours and may be necessary on a weekly basis. Some systems require that this function be done **off-line**, with the file system unmounted. During this **down time**, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines. Most modern systems that need defragmentation can perform it **on-line** during normal system operations, but the performance penalty can be substantial.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example). In general, however, the size of an output file may be difficult to estimate.

If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming. The user need never be informed explicitly about what is happening, however; the system continues despite the problem, although more and more slowly.

Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that will grow slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of internal fragmentation.

To minimize these drawbacks, an operating system can use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially. Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal

fragmentation can still be a problem if the extents are too large, and external fragmentation can become a problem as extents of varying sizes are allocated and deallocated. The commercial Symantec Veritas file system uses extents to optimize performance. Veritas is a high-performance replacement for the standard UNIX UFS.

14.4.2 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of storage blocks; the blocks may be scattered anywhere on the device. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (Figure 14.5). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a block address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first block of the file. This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the *i*th

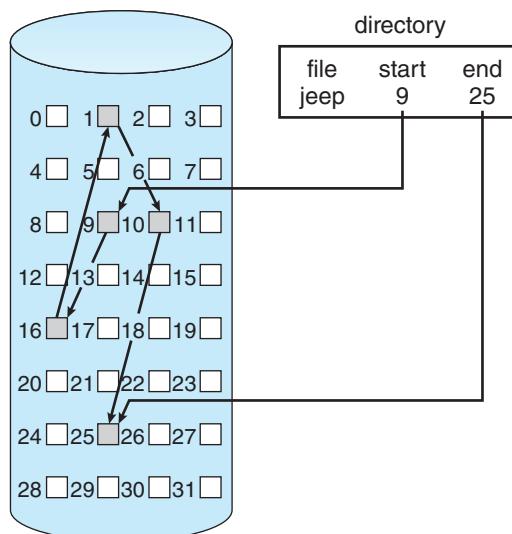


Figure 14.5 Linked allocation of disk space.

block of a file, we must start at the beginning of that file and follow the pointers until we get to the i th block. Each access to a pointer requires a storage device read, and some require an HDD seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the secondary storage device only in cluster units. Pointers then use a much smaller percentage of the file's space. This method allows the logical-to-physical block mapping to remain simple but improves HDD throughput (because fewer disk-head seeks are required) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full. Also random I/O performance suffers because a request for a small amount of data transfers a large amount of data. Clusters can be used to improve the disk-access time for many other algorithms as well, so they are used in most file systems.

Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the device, and consider what would happen if a pointer was lost or damaged. A bug in the operating-system software or a hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block. However, these schemes require even more overhead for each file.

An important variation on linked allocation is the use of a **file-allocation table (FAT)**. This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of storage at the beginning of each volume is set aside to contain the table. The table has one entry for each block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 14.6 for a file consisting of disk blocks 217, 618, and 339.

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

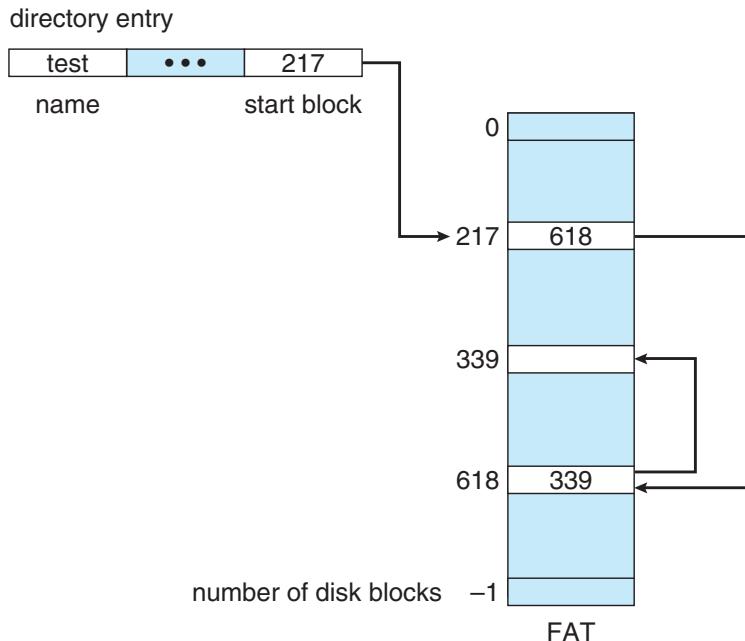


Figure 14.6 File-allocation table.

14.4.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

Each file has its own index block, which is an array of storage-block addresses. The i th entry in the index block points to the i th block of the file. The directory contains the address of the index block (Figure 14.7). To find and read the i th block, we use the pointer in the i th index-block entry. This scheme is similar to the paging scheme described in Section 9.3.

When the file is created, all pointers in the index block are set to null. When the i th block is first written, a block is obtained from the free-space manager, and its address is put in the i th index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the storage device can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as

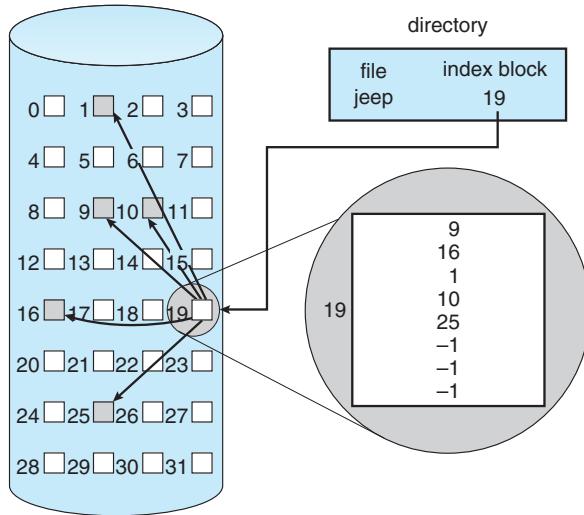


Figure 14.7 Indexed allocation of disk space.

possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

- **Linked scheme.** An index block is normally one storage block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is null (for a small file) or is a pointer to another index block (for a large file).
- **Multilevel index.** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.
- **Combined scheme.** Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to **indirect blocks**. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**. (A UNIX inode is shown in Figure 14.8.)

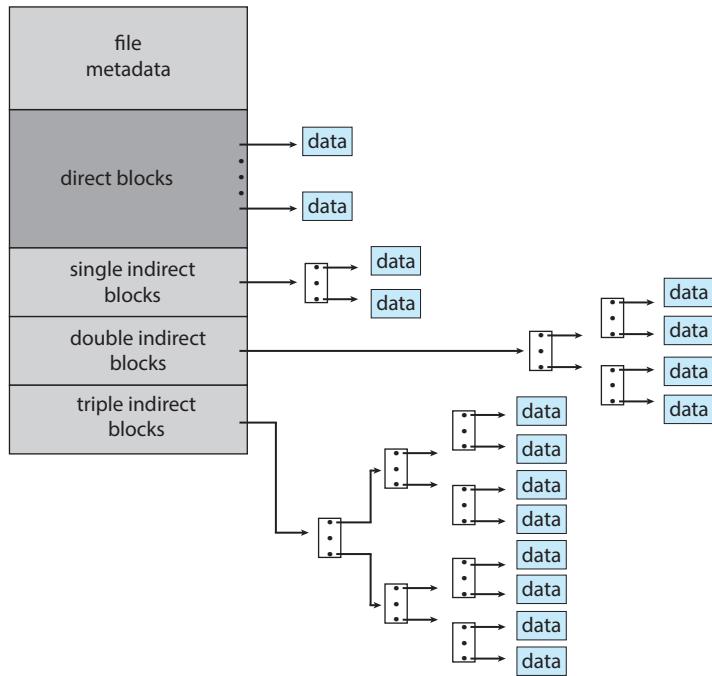


Figure 14.8 The UNIX inode.

Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems. A 32-bit file pointer reaches only 2^{32} bytes, or 4 GB. Many UNIX and Linux implementations now support 64-bit file pointers, which allows files and file systems to be several exabytes in size. The ZFS file system supports 128-bit file pointers.

Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a volume.

14.4.4 Performance

The allocation methods that we have discussed vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper method or methods for an operating system to implement.

Before selecting an allocation method, we need to determine how the systems will be used. A system with mostly sequential access should not use the same method as a system with mostly random access.

For any type of access, contiguous allocation requires only one access to get a block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the address of the i th block (or the next block) and read it directly.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the i th block might require i block reads. This

problem indicates why linked allocation should not be used for an application requiring direct access.

As a result, some systems support direct-access files by using contiguous allocation and sequential-access files by using linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created. In this case, the operating system must have appropriate data structures and algorithms to support both allocation methods. Files can be converted from one type to another by the creation of a new file of the desired type, into which the contents of the old file are copied. The old file may then be deleted and the new file renamed.

Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks before the needed data block finally could be read. Thus, the performance of indexed allocation depends on the index structure, on the size of the file, and on the position of the block desired.

Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks) and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

Many other optimizations are in use. Given the disparity between CPU speed and disk speed, it is not unreasonable to add thousands of extra instructions to the operating system to save just a few disk-head movements. Furthermore, this disparity is increasing over time, to the point where hundreds of thousands of instructions could reasonably be used to optimize head movements.

For NVM devices, there are no disk head seeks, so different algorithms and optimizations are needed. Using an old algorithm that spends many CPU cycles trying to avoid a nonexistent head movement would be very inefficient. Existing file systems are being modified and new ones being created to attain maximum performance from NVM storage devices. These developments aim to reduce the instruction count and overall path between the storage device and application access to the data.

14.5 Free-Space Management

Since storage space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks allow only one write to any given sector, and thus reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free device blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate

that space to the new file. This space is then removed from the free-space list. When a file is deleted, its space is added to the free-space list. The free-space list, despite its name, is not necessarily implemented as a list, as we discuss next.

14.5.1 Bit Vector

Frequently, the free-space list is implemented as a **bitmap** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

00111100111110001100000011100000 ...

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. One technique for finding the first free block on a system that uses a bit vector to allocate space is to sequentially check each word in the bitmap to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

$$(\text{number of bits per word}) \times (\text{number of 0-value words}) + \text{offset of first 1 bit.}$$

Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to the device containing the file system occasionally for recovery needs). Keeping it in main memory is possible for smaller devices but not necessarily for larger ones. A 1.3-GB disk with 512-byte blocks would need a bitmap of over 332 KB to track its free blocks, although clustering the blocks in groups of four reduces this number to around 83 KB per disk. A 1-TB disk with 4-KB blocks would require 32 MB ($2^{40} / 2^{12} = 2^{28}$ bits = 2^{25} bytes = 2^5 MB) to store its bitmap. Given that disk size constantly increases, the problem with bit vectors will continue to escalate as well.

14.5.2 Linked List

Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory. This first block contains a pointer to the next free block, and so on. Recall our earlier example (Section 14.5.1), in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 14.9). This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time on HDDs. Fortunately, however, traversing the free list is not a frequent action. Usually,

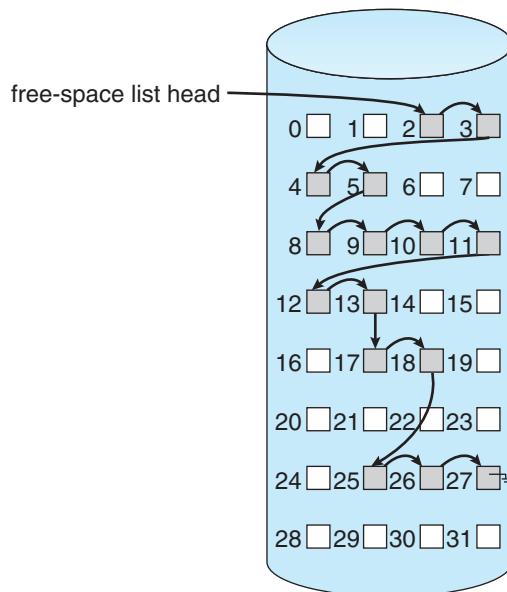


Figure 14.9 Linked free-space list on disk.

the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

14.5.3 Grouping

A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first $n-1$ of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

14.5.4 Counting

Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free block addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a device address and a count. Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than 1. Note that this method of tracking free space is similar to the extent method of allocating blocks. These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

14.5.5 Space Maps

Oracle's **ZFS** file system (found in Solaris and some other operating systems) was designed to encompass huge numbers of files, directories, and even file systems (in ZFS, we can create file-system hierarchies). On these scales, metadata I/O can have a large performance impact. Consider, for example, that if the free-space list is implemented as a bitmap, bitmaps must be modified both when blocks are allocated and when they are freed. Freeing 1 GB of data on a 1-TB disk could cause thousands of blocks of bitmaps to be updated, because those data blocks could be scattered over the entire disk. Clearly, the data structures for such a system could be large and inefficient.

In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures. First, ZFS creates **metaslabs** to divide the space on the device into chunks of manageable size. A given volume may contain hundreds of metaslabs. Each metaslab has an associated space map. ZFS uses the counting algorithm to store information about free blocks. Rather than write counting structures to disk, it uses log-structured file-system techniques to record them. The space map is a log of all block activity (allocating and freeing), in time order, in counting format. When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the log into that structure. The in-memory space map is then an accurate representation of the allocated and free space in the metaslab. ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry. Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS. During the collection and sorting phase, block requests can still occur, and ZFS satisfies these requests from the log. In essence, the log plus the balanced tree *is* the free list.

14.5.6 TRIMming Unused Blocks

HDDs and other storage media that allow blocks to be overwritten for updates need only the free list for managing free space. Blocks do not need to be treated specially when freed. A freed block typically keeps its data (but without any file pointers to the block) until the data are overwritten when the block is next allocated.

Storage devices that do not allow overwrite, such as NVM flash-based storage devices, suffer badly when these same algorithms are applied. Recall from Section 11.1.2 that such devices must be erased before they can again be written to, and that those erases must be made in large chunks (blocks, composed of pages) and take a relatively long time compared with reads or writes.

A new mechanism is needed to allow the file system to inform the storage device that a page is free and can be considered for erasure (once the block containing the page is entirely free). That mechanism varies based on the storage controller. For ATA-attached drives, it is TRIM, while for NVMe-based storage, it is the unallocate command. Whatever the specific controller command, this mechanism keeps storage space available for writing. Without such a capability, the storage device gets full and needs garbage collection and block erasure, leading to decreases in storage I/O write performance (known as "a write cliff").

With the TRIM mechanism and similar capabilities, the garbage collection and erase steps can occur before the device is nearly full, allowing the device to provide more consistent performance.

14.6 Efficiency and Performance

Now that we have discussed various block-allocation and directory-management options, we can further consider their effect on performance and efficient storage use. Disks tend to represent a major bottleneck in system performance, since they are the slowest main computer component. Even NVM devices are slow compared with CPU and main memory, so their performance must be optimized as well. In this section, we discuss a variety of techniques used to improve the efficiency and performance of secondary storage.

14.6.1 Efficiency

The efficient use of storage device space depends heavily on the allocation and directory algorithms in use. For instance, UNIX inodes are preallocated on a volume. Even an empty disk has a percentage of its space lost to inodes. However, by preallocating the inodes and spreading them across the volume, we improve the file system's performance. This improved performance results from the UNIX allocation and free-space algorithms, which try to keep a file's data blocks near that file's inode block to reduce seek time.

As another example, let's reconsider the clustering scheme discussed in Section 14.4, which improves file-seek and file-transfer performance at the cost of internal fragmentation. To reduce this fragmentation, BSD UNIX varies the cluster size as a file grows. Large clusters are used where they can be filled, and small clusters are used for small files and the last cluster of a file. This system is described in Appendix C.

The types of data normally kept in a file's directory (or inode) entry also require consideration. Commonly, a "last write date" is recorded to supply information to the user and to determine whether the file needs to be backed up. Some systems also keep a "last access date," so that a user can determine when the file was last read. The result of keeping this information is that, whenever the file is read, a field in the directory structure must be written to. That means the block must be read into memory, a section changed, and the block written back out to the device, because operations on secondary storage occur only in block (or cluster) chunks. So any time a file is opened for reading, its FCB must be read and written as well. This requirement can be inefficient for frequently accessed files, so we must weigh its benefit against its performance cost when designing a file system. Generally, every data item associated with a file needs to be considered for its effect on efficiency and performance.

Consider, for instance, how efficiency is affected by the size of the pointers used to access data. Most systems use either 32-bit or 64-bit pointers throughout the operating system. Using 32-bit pointers limits the size of a file to 2^{32} , or 4 GB. Using 64-bit pointers allows very large file sizes, but 64-bit pointers require more space to store. As a result, the allocation and free-space-management methods (linked lists, indexes, and so on) use more storage space.

One of the difficulties in choosing a pointer size—or, indeed, any fixed allocation size within an operating system—is planning for the effects of changing technology. Consider that the IBM PC XT had a 10-MB hard drive and an MS-DOS FAT file system that could support only 32 MB. (Each FAT entry was 12 bits, pointing to an 8-KB cluster.) As disk capacities increased, larger disks had to be split into 32-MB partitions, because the file system could not track blocks beyond 32 MB. As hard disks with capacities of over 100 MB became common, the disk data structures and algorithms in MS-DOS had to be modified to allow larger file systems. (Each FAT entry was expanded to 16 bits and later to 32 bits.) The initial file-system decisions were made for efficiency reasons; however, with the advent of MS-DOS Version 4, millions of computer users were inconvenienced when they had to switch to the new, larger file system. Solaris's ZFS file system uses 128-bit pointers, which theoretically should never need to be extended. (The minimum mass of a device capable of storing 2^{128} bytes using atomic-level storage would be about 272 trillion kilograms.)

As another example, consider the evolution of the Solaris operating system. Originally, many data structures were of fixed length, allocated at system startup. These structures included the process table and the open-file table. When the process table became full, no more processes could be created. When the file table became full, no more files could be opened. The system would fail to provide services to users. Table sizes could be increased only by recompiling the kernel and rebooting the system. With later releases of Solaris, (as with modern Linux kernels) almost all kernel structures were allocated dynamically, eliminating these artificial limits on system performance. Of course, the algorithms that manipulate these tables are more complicated, and the operating system is a little slower because it must dynamically allocate and deallocate table entries; but that price is the usual one for more general functionality.

14.6.2 Performance

Even after the basic file-system algorithms have been selected, we can still improve performance in several ways. As was discussed in Chapter 12, storage device controllers include local memory to form an on-board cache that is large enough to store entire tracks or blocks at a time. On an HDD, once a seek is performed, the track is read into the disk cache starting at the sector under the disk head (reducing latency time). The disk controller then transfers any sector requests to the operating system. Once blocks make it from the disk controller into main memory, the operating system may cache the blocks there.

Some systems maintain a separate section of main memory for a **buffer cache**, where blocks are kept under the assumption that they will be used again shortly. Other systems cache file data using a **page cache**. The page cache uses virtual memory techniques to cache file data as pages rather than as file-system-oriented blocks. Caching file data using virtual addresses is far more efficient than caching through physical disk blocks, as accesses interface with virtual memory rather than the file system. Several systems—including Solaris, Linux, and Windows—use page caching to cache both process pages and file data. This is known as **unified virtual memory**.

Some versions of UNIX and Linux provide a **unified buffer cache**. To illustrate the benefits of the unified buffer cache, consider the two alternatives

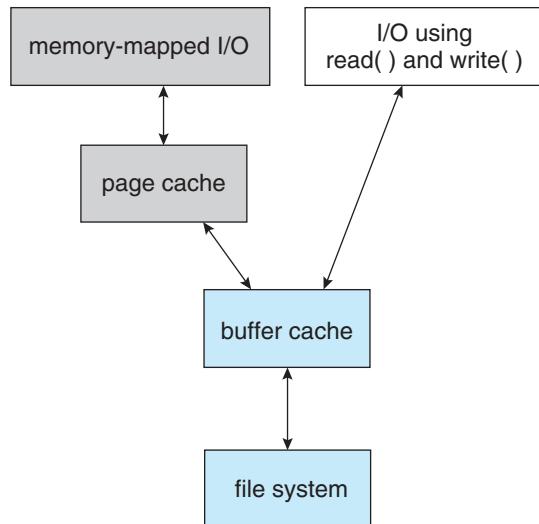


Figure 14.10 I/O without a unified buffer cache.

for opening and accessing a file. One approach is to use memory mapping (Section 13.5); the second is to use the standard system calls `read()` and `write()`. Without a unified buffer cache, we have a situation similar to Figure 14.10. Here, the `read()` and `write()` system calls go through the buffer cache. The memory-mapping call, however, requires using two caches—the page cache and the buffer cache. A memory mapping proceeds by reading in disk blocks from the file system and storing them in the buffer cache. Because the virtual memory system does not interface with the buffer cache, the contents of the file in the buffer cache must be copied into the page cache. This situation, known as **double caching**, requires caching file-system data twice. Not only does this waste memory but it also wastes significant CPU and I/O cycles due to the extra data movement within system memory. In addition, inconsistencies between the two caches can result in corrupt files. In contrast, when a unified buffer cache is provided, both memory mapping and the `read()` and `write()` system calls use the same page cache. This has the benefit of avoiding double caching, and it allows the virtual memory system to manage file-system data. The unified buffer cache is shown in Figure 14.11.

Regardless of whether we are caching storage blocks or pages (or both), least recently used (LRU) (Section 10.4.4) seems a reasonable general-purpose algorithm for block or page replacement. However, the evolution of the Solaris page-caching algorithms reveals the difficulty in choosing an algorithm. Solaris allows processes and the page cache to share unused memory. Versions earlier than Solaris 2.5.1 made no distinction between allocating pages to a process and allocating them to the page cache. As a result, a system performing many I/O operations used most of the available memory for caching pages. Because of the high rates of I/O, the page scanner (Section 10.10.3) reclaimed pages from processes—rather than from the page cache—when free memory ran low. Solaris 2.6 and Solaris 7 optionally implemented priority paging, in which the page scanner gave priority to process pages over the page cache. Solaris 8 applied a fixed limit to process pages and the file-system page cache, prevent-

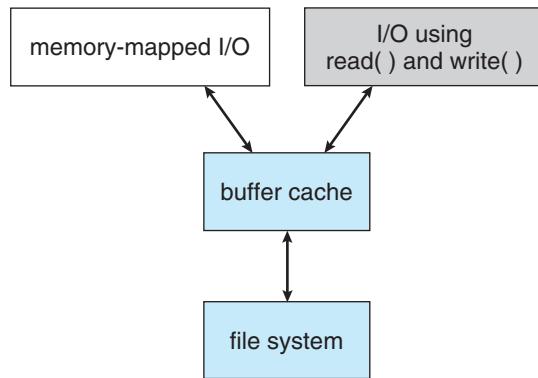


Figure 14.11 I/O using a unified buffer cache.

ing either from forcing the other out of memory. Solaris 9 and 10 again changed the algorithms to maximize memory use and minimize thrashing.

Another issue that can affect the performance of I/O is whether writes to the file system occur synchronously or asynchronously. **Synchronous writes** occur in the order in which the storage subsystem receives them, and the writes are not buffered. Thus, the calling routine must wait for the data to reach the drive before it can proceed. In an **asynchronous write**, the data are stored in the cache, and control returns to the caller. Most writes are asynchronous. However, metadata writes, among others, can be synchronous. Operating systems frequently include a flag in the open system call to allow a process to request that writes be performed synchronously. For example, databases use this feature for atomic transactions, to assure that data reach stable storage in the required order.

Some systems optimize their page cache by using different replacement algorithms, depending on the access type of the file. A file being read or written sequentially should not have its pages replaced in LRU order, because the most recently used page will be used last, or perhaps never again. Instead, sequential access can be optimized by techniques known as free-behind and read-ahead. **Free-behind** removes a page from the buffer as soon as the next page is requested. The previous pages are not likely to be used again and waste buffer space. With **read-ahead**, a requested page and several subsequent pages are read and cached. These pages are likely to be requested after the current page is processed. Retrieving these data from the disk in one transfer and caching them saves a considerable amount of time. One might think that a track cache on the controller would eliminate the need for read-ahead on a multiprogrammed system. However, because of the high latency and overhead involved in making many small transfers from the track cache to main memory, performing a read-ahead remains beneficial.

The page cache, the file system, and the device drivers have some interesting interactions. When small amounts of data are written to a file, the pages are buffered in the cache, and the storage device driver sorts its output queue according to device address. These two actions allow a disk driver to minimize disk-head seeks. Unless synchronous writes are required, a process writing to disk simply writes into the cache, and the system asynchronously writes the

data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much more nearly asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for small transfers, counter to intuition. No matter how much buffering and caching is available, large, continuous I/O can overrun the capacity and end up bottlenecked on the device's performance. Consider writing a large movie file to a HDD. If the file is larger than the page cache (or the part of the page cache available to the process) then the page cache will fill and all I/O will occur at drive speed. Current HDDs read faster than they write, so in this instance the performance aspects are reversed from smaller I/O performance.

14.7 Recovery

Files and directories are kept both in main memory and on the storage volume, and care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency. A system crash can cause inconsistencies among on-storage file-system data structures, such as directory structures, free-block pointers, and free FCB pointers. Many file systems apply changes to these structures in place. A typical operation, such as creating a file, can involve many structural changes within the file system on the disk. Directory structures are modified, FCBs are allocated, data blocks are allocated, and the free counts for all of these blocks are decreased. These changes can be interrupted by a crash, and inconsistencies among the structures can result. For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB. Compounding this problem is the caching that operating systems do to optimize I/O performance. Some changes may go directly to storage, while others may be cached. If the cached changes do not reach the storage device before a crash occurs, more corruption is possible.

In addition to crashes, bugs in file-system implementation, device controllers, and even user applications can corrupt a file system. File systems have varying methods to deal with corruption, depending on the file-system data structures and algorithms. We deal with these issues next.

14.7.1 Consistency Checking

Whatever the cause of corruption, a file system must first detect the problems and then correct them. For detection, a scan of all the metadata on each file system can confirm or deny the consistency of the system. Unfortunately, this scan can take minutes or hours and should occur every time the system boots. Alternatively, a file system can record its state within the file-system metadata. At the start of any metadata change, a status bit is set to indicate that the metadata is in flux. If all updates to the metadata complete successfully, the file system can clear that bit. If, however, the status bit remains set, a consistency checker is run.

The **consistency checker**—a systems program such as `fsck` in UNIX—compares the data in the directory structure and other metadata with the state on storage and tries to fix any inconsistencies it finds. The allocation and free-space-management algorithms dictate what types of problems the

checker can find and how successful it will be in fixing them. For instance, if linked allocation is used and there is a link from any block to its next block, then the entire file can be reconstructed from the data blocks, and the directory structure can be recreated. In contrast, the loss of a directory entry on an indexed allocation system can be disastrous, because the data blocks have no knowledge of one another. For this reason, some UNIX file systems cache directory entries for reads, but any write that results in space allocation, or other metadata changes, is done synchronously, before the corresponding data blocks are written. Of course, problems can still occur if a synchronous write is interrupted by a crash. Some NVM storage devices contain a battery or supercapacitor to provide enough power, even during a power loss, to write data from device buffers to the storage media so the data are not lost. But even those precautions do not protect against corruption due to a crash.

14.7.2 Log-Structured File Systems

Computer scientists often find that algorithms and technologies originally used in one area are equally useful in other areas. Such is the case with the database log-based recovery algorithms. These logging algorithms have been applied successfully to the problem of consistency checking. The resulting implementations are known as **log-based transaction-oriented** (or **journaling**) file systems.

Note that with the consistency-checking approach discussed in the preceding section, we essentially allow structures to break and repair them on recovery. However, there are several problems with this approach. One is that the inconsistency may be irreparable. The consistency check may not be able to recover the structures, resulting in loss of files and even entire directories. Consistency checking can require human intervention to resolve conflicts, and that is inconvenient if no human is available. The system can remain unavailable until the human tells it how to proceed. Consistency checking also takes system and clock time. To check terabytes of data, hours of clock time may be required.

The solution to this problem is to apply log-based recovery techniques to file-system metadata updates. Both NTFS and the Veritas file system use this method, and it is included in recent versions of UFS on Solaris. In fact, it is now common on many file systems including ext3, ext4, and ZFS.

Fundamentally, all metadata changes are written sequentially to a log. Each set of operations for performing a specific task is a **transaction**. Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution. Meanwhile, these log entries are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, an entry is made in the log indicating that. The log file is actually a circular buffer. A **circular buffer** writes to the end of its space and then continues at the beginning, overwriting older values as it goes. We would not want the buffer to write over data that had not yet been saved, so that scenario is avoided. The log may be in a separate section of the file system or even on a separate storage device.

If the system crashes, the log file will contain zero or more transactions. Any transactions it contains were not completed to the file system, even though they were committed by the operating system, so they must now be completed. The transactions can be executed from the pointer until the work is complete so that the file-system structures remain consistent. The only problem occurs when a transaction was aborted—that is, was not committed before the system crashed. Any changes from such a transaction that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating any problems with consistency checking.

A side benefit of using logging on disk metadata updates is that those updates proceed much faster than when they are applied directly to the on-disk data structures. The reason is found in the performance advantage of sequential I/O over random I/O. The costly synchronous random metadata writes are turned into much less costly synchronous sequential writes to the log-structured file system's logging area. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of metadata-oriented operations, such as file creation and deletion, on HDD storage.

14.7.3 Other Solutions

Another alternative to consistency checking is employed by Network Appliance's WAFL file system and the Solaris ZFS file system. These systems never overwrite blocks with new data. Rather, a transaction writes all data and metadata changes to new blocks. When the transaction is complete, the metadata structures that pointed to the old versions of these blocks are updated to point to the new blocks. The file system can then remove the old pointers and the old blocks and make them available for reuse. If the old pointers and blocks are kept, a **snapshot** is created; the snapshot is a view of the file system at a specific point in time (before any updates after that time were applied). This solution should require no consistency checking if the pointer update is done atomically. WAFL does have a consistency checker, however, so some failure scenarios can still cause metadata corruption. (See Section 14.8 for details of the WAFL file system.)

ZFS takes an even more innovative approach to disk consistency. Like WAFL, it never overwrites blocks. However, ZFS goes further and provides checksumming of all metadata and data blocks. This solution (when combined with RAID) assures that data are always correct. ZFS therefore has no consistency checker. (More details on ZFS are found in Section 11.8.6.)

14.7.4 Backup and Restore

Storage devices sometimes fail, and care must be taken to ensure that the data lost in such a failure are not lost forever. To this end, system programs can be used to **back up** data from one storage device to another, such as a magnetic tape or other secondary storage device. Recovery from the loss of an individual file, or of an entire device, may then be a matter of **restoring** the data from backup.

To minimize the copying needed, we can use information from each file's directory entry. For instance, if the backup program knows when the last

backup of a file was done, and the file's last write date in the directory indicates that the file has not changed since that date, then the file does not need to be copied again. A typical backup schedule may then be as follows:

- **Day 1.** Copy to a backup medium all files from the file system. This is called a **full backup**.
 - **Day 2.** Copy to another medium all files changed since day 1. This is an **incremental backup**.
 - **Day 3.** Copy to another medium all files changed since day 2.
- ...
- **Day N .** Copy to another medium all files changed since day $N - 1$. Then go back to day 1.

The new cycle can have its backup written over the previous set or onto a new set of backup media.

Using this method, we can restore an entire file system by starting restores with the full backup and continuing through each of the incremental backups. Of course, the larger the value of N , the greater the number of media that must be read for a complete restore. An added advantage of this backup cycle is that we can restore any file accidentally deleted during the cycle by retrieving the deleted file from the backup of the previous day.

The length of the cycle is a compromise between the amount of backup needed and the number of days covered by a restore. To decrease the number of tapes that must be read to do a restore, an option is to perform a full backup and then each day back up all files that have changed since the full backup. In this way, a restore can be done via the most recent incremental backup and the full backup, with no other incremental backups needed. The trade-off is that more files will be modified each day, so each successive incremental backup involves more files and more backup media.

A user may notice that a particular file is missing or corrupted long after the damage was done. For this reason, we usually plan to take a full backup from time to time that will be saved "forever." It is a good idea to store these permanent backups far away from the regular backups to protect against hazard, such as a fire that destroys the computer and all the backups too. In the TV show "Mr. Robot," hackers not only attacked the primary sources of banks' data but also their backup sites. Having multiple backup sites might not be a bad idea if your data are important.

14.8 Example: The WAFL File System

Because secondary-storage I/O has such a huge impact on system performance, file-system design and implementation command quite a lot of attention from system designers. Some file systems are general purpose, in that they can provide reasonable performance and functionality for a wide variety of file sizes, file types, and I/O loads. Others are optimized for specific tasks in an attempt to provide better performance in those areas than general-purpose

file systems. The [write-anywhere file layout \(WAFL\)](#) from NetApp, Inc. is an example of this sort of optimization. WAFL is a powerful, elegant file system optimized for random writes.

WAFL is used exclusively on network file servers produced by NetApp and is meant for use as a distributed file system. It can provide files to clients via the NFS, CIFS, iSCSI, ftp, and http protocols, although it was designed just for NFS and CIFS. When many clients use these protocols to talk to a file server, the server may see a very large demand for random reads and an even larger demand for random writes. The NFS and CIFS protocols cache data from read operations, so writes are of the greatest concern to file-server creators.

WAFL is used on file servers that include an NVRAM cache for writes. The WAFL designers took advantage of running on a specific architecture to optimize the file system for random I/O, with a stable-storage cache in front. Ease of use is one of the guiding principles of WAFL. Its creators also designed it to include a new snapshot functionality that creates multiple read-only copies of the file system at different points in time, as we shall see.

The file system is similar to the Berkeley Fast File System, with many modifications. It is block-based and uses inodes to describe files. Each inode contains 16 pointers to blocks (or indirect blocks) belonging to the file described by the inode. Each file system has a root inode. All of the metadata lives in files. All inodes are in one file, the free-block map in another, and the free-inode map in a third, as shown in Figure 14.12. Because these are standard files, the data blocks are not limited in location and can be placed anywhere. If a file system is expanded by addition of disks, the lengths of the metadata files are automatically expanded by the file system.

Thus, a WAFL file system is a tree of blocks with the root inode as its base. To take a snapshot, WAFL creates a copy of the root inode. Any file or metadata updates after that go to new blocks rather than overwriting their existing blocks. The new root inode points to metadata and data changed as a result of these writes. Meanwhile, the snapshot (the old root inode) still points to the old blocks, which have not been updated. It therefore provides access to the file system just as it was at the instant the snapshot was made—and takes very little storage space to do so. In essence, the extra space occupied by a snapshot consists of just the blocks that have been modified since the snapshot was taken.

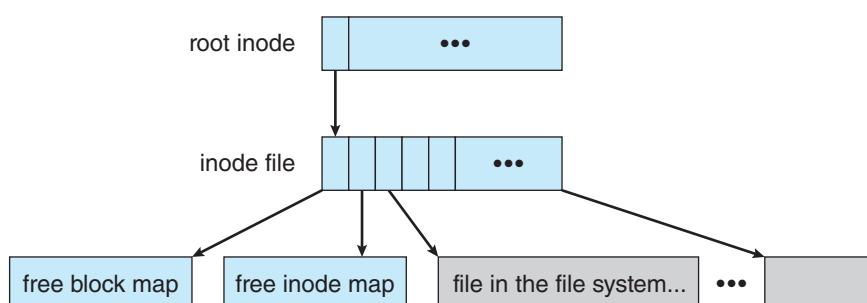


Figure 14.12 The WAFL file layout.

An important change from more standard file systems is that the free-block map has more than one bit per block. It is a bitmap with a bit set for each snapshot that is using the block. When all snapshots that have been using the block are deleted, the bitmap for that block is all zeros, and the block is free to be reused. Used blocks are never overwritten, so writes are very fast, because a write can occur at the free block nearest the current head location. There are many other performance optimizations in WAFL as well.

Many snapshots can exist simultaneously, so one can be taken each hour of the day and each day of the month, for example. A user with access to these snapshots can access files as they were at any of the times the snapshots were taken. The snapshot facility is also useful for backups, testing, versioning, and so on. WAFL's snapshot facility is very efficient in that it does not even require that copy-on-write copies of each data block be taken before the block is modified. Other file systems provide snapshots, but frequently with less efficiency. WAFL snapshots are depicted in Figure 14.13.

Newer versions of WAFL actually allow read–write snapshots, known as **clones**. Clones are also efficient, using the same techniques as snapshots. In this case, a read-only snapshot captures the state of the file system, and a clone refers back to that read-only snapshot. Any writes to the clone are stored in new blocks, and the clone's pointers are updated to refer to the new blocks. The original snapshot is unmodified, still giving a view into the file system as

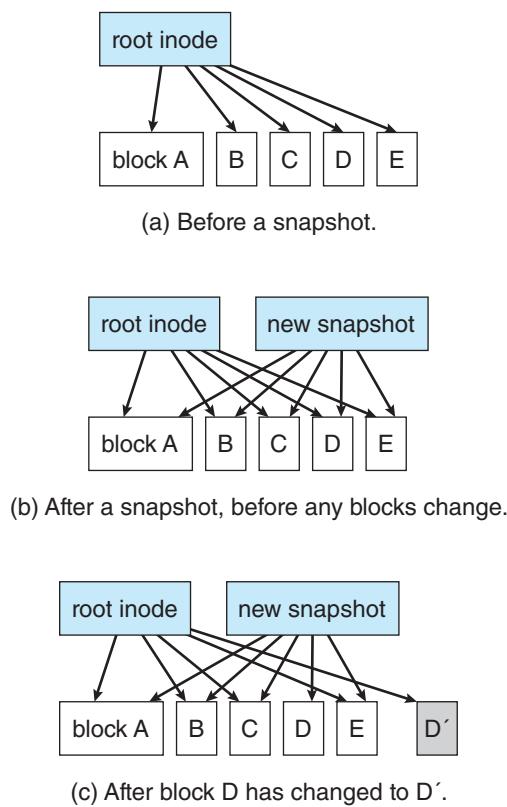


Figure 14.13 Snapshots in WAFL.

THE APPLE FILE SYSTEM

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

Apple File System (APFS) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, iOS, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is available as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). **Fast directory sizing** provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

it was before the clone was updated. Clones can also be promoted to replace the original file system; this involves throwing out all of the old pointers and any associated old blocks. Clones are useful for testing and upgrades, as the original version is left untouched and the clone deleted when the test is done or if the upgrade fails.

Another feature that naturally results from the WAFL file system implementation is **replication**, the duplication and synchronization of a set of data over a network to another system. First, a snapshot of a WAFL file system is duplicated to another system. When another snapshot is taken on the source system, it is relatively easy to update the remote system just by sending over all blocks contained in the new snapshot. These blocks are the ones that have changed between the times the two snapshots were taken. The remote system adds these blocks to the file system and updates its pointers, and the new system then is a duplicate of the source system as of the time of the second snapshot. Repeating this process maintains the remote system as a nearly up-to-date copy of the first system. Such replication is used for disaster recovery. Should the first system be destroyed, most of its data are available for use on the remote system.

Finally, note that the ZFS file system supports similarly efficient snapshots, clones, and replication, and those features are becoming more common in various file systems as time goes by.

14.9 Summary

- Most file systems reside on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk, but the use of NVM devices is increasing.
- Storage devices are segmented into partitions to control media use and to allow multiple, possibly varying, file systems on a single device. These file systems are mounted onto a logical file system architecture to make them available for use.
- File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices and communicating with them. Upper levels deal with symbolic file names and logical properties of files.
- The various files within a file system can be allocated space on the storage device in three ways: through contiguous, linked, or indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block. These algorithms can be optimized in many ways. Contiguous space can be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.
- Free-space allocation methods also influence the efficiency of disk-space use, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.
- Directory-management routines must consider efficiency, performance, and reliability. A hash table is a commonly used method, as it is fast and efficient. Unfortunately, damage to the table or a system crash can result in inconsistency between the directory information and the disk's contents.
- A consistency checker can be used to repair damaged file-system structures. Operating-system backup tools allow data to be copied to magnetic tape or other storage devices, enabling the user to recover from data loss or even entire device loss due to hardware failure, operating system bug, or user error.
- Due to the fundamental role that file systems play in system operation, their performance and reliability are crucial. Techniques such as log structures and caching help improve performance, while log structures and RAID improve reliability. The WAFL file system is an example of optimization of performance to match a specific I/O load.

Practice Exercises

- 14.1** Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grow at the end. Also assume that the block information to be added is stored in memory.
- The block is added at the beginning.
 - The block is added in the middle.
 - The block is added at the end.
 - The block is removed from the beginning.
 - The block is removed from the middle.
 - The block is removed from the end.
- 14.2** Why must the bit map for file allocation be kept on mass storage, rather than in main memory?
- 14.3** Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?
- 14.4** One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area of a specified size. If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.
- 14.5** How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?
- 14.6** Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

Further Reading

The internals of the BSD UNIX system are covered in full in [McKusick et al. (2015)]. Details concerning file systems for Linux can be found in [Love (2010)]. The Google file system is described in [Ghemawat et al. (2003)]. FUSE can be found at <http://fuse.sourceforge.net>.

Log-structured file organizations for enhancing both performance and consistency are discussed in [Rosenblum and Ousterhout (1991)], [Seltzer et al. (1993)], and [Seltzer et al. (1995)]. Log-structured designs for networked file systems are proposed in [Hartman and Ousterhout (1995)] and [Thekkath et al. (1997)].

The ZFS source code for space maps can be found at http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/fs/zfs/space_map.c.

ZFS documentation can be found at <http://www.opensolaris.org/os/community/ZFS/docs>.

The NTFS file system is explained in [Solomon (1998)], the Ext3 file system used in Linux is described in [Mauerer (2008)], and the WAFL file system is covered in [Hitz et al. (1995)].

Bibliography

- [Ghemawat et al. (2003)] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System”, *Proceedings of the ACM Symposium on Operating Systems Principles* (2003).
- [Hartman and Ousterhout (1995)] J. H. Hartman and J. K. Ousterhout, “The Zebra Striped Network File System”, *ACM Transactions on Computer Systems*, Volume 13, Number 3 (1995), pages 274–310.
- [Hitz et al. (1995)] D. Hitz, J. Lau, and M. Malcolm, “File System Design for an NFS File Server Appliance”, Technical report, NetApp (1995).
- [Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).
- [Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).
- [McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System—Second Edition*, Pearson (2015).
- [Rosenblum and Ousterhout (1991)] M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), pages 1–15.
- [Seltzer et al. (1993)] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, “An Implementation of a Log-Structured File System for UNIX”, *USENIX Winter* (1993), pages 307–326.
- [Seltzer et al. (1995)] M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. N. Padmanabhan, “File System Logging Versus Clustering: A Performance Comparison”, *USENIX Winter* (1995), pages 249–264.
- [Solomon (1998)] D. A. Solomon, *Inside Windows NT*, Second Edition, Microsoft Press (1998).

[Thekkath et al. (1997)] C. A. Thekkath, T. Mann, and E. K. Lee, “Frangipani: A Scalable Distributed File System”, *Symposium on Operating Systems Principles* (1997), pages 224–237.

Chapter 14 Exercises

- 14.7** Consider a file system that uses a modified contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?
- All extents are of the same size, and the size is predetermined.
 - Extents can be of any size and are allocated dynamically.
 - Extents can be of a few fixed sizes, and these sizes are predetermined.
- 14.8** Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.
- 14.9** What are the advantages of the variant of linked allocation that uses a FAT to chain together the blocks of a file?
- 14.10** Consider a system where free space is kept in a free-space list.
- Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.
 - Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at /a/b/c? Assume that none of the disk blocks is currently being cached.
 - Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.
- 14.11** Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4 KB of disk space as a single 4-KB block or as eight 512-byte blocks. How could we take advantage of this flexibility to improve performance? What modifications would have to be made to the free-space management scheme in order to support this feature?
- 14.12** Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.
- 14.13** Discuss the advantages and disadvantages of supporting links to files that cross mount points (that is, the file link refers to a file that is stored in a different volume).
- 14.14** Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

- a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)
 - b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?
- 14.15** Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?
- 14.16** Fragmentation on a storage device can be eliminated through compaction. Typical disk devices do not have relocation or base registers (such as those used when memory is to be compacted), so how can we relocate files? Give three reasons why compacting and relocating files are often avoided.
- 14.17** Explain why logging metadata updates ensures recovery of a file system after a file-system crash.
- 14.18** Consider the following backup scheme:
- **Day 1.** Copy to a backup medium all files from the disk.
 - **Day 2.** Copy to another medium all files changed since day 1.
 - **Day 3.** Copy to another medium all files changed since day 1.
- This differs from the schedule given in Section 14.7.4 by having all subsequent backups copy all files modified since the first full backup. What are the benefits of this system over the one in Section 14.7.4? What are the drawbacks? Are restore operations made easier or more difficult? Explain your answer.
- 14.19** Discuss the advantages and disadvantages of associating with remote file systems (stored on file servers) a set of failure semantics different from those associated with local file systems.
- 14.20** What are the implications of supporting UNIX consistency semantics for shared access to files stored on remote file systems?

File-System Internals



As we saw in Chapter 13, the file system provides the mechanism for on-line storage and access to file contents, including data and programs. This chapter is primarily concerned with the internal structures and operations of file systems. We explore in detail ways to structure file use, to allocate storage space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage.

CHAPTER OBJECTIVES

- Delve into the details of file systems and their implementation.
- Explore booting and file sharing.
- Describe remote file systems, using NFS as an example.

15.1 File Systems

Certainly, no general-purpose computer stores just one file. There are typically thousands, millions, even billions of files within a computer. Files are stored on random-access storage devices, including hard disk drives, optical disks, and nonvolatile memory devices.

As you have seen in the preceding chapters, a general-purpose computer system can have multiple storage devices, and those devices can be sliced up into partitions, which hold volumes, which in turn hold file systems. Depending on the volume manager, a volume may span multiple partitions as well. Figure 15.1 shows a typical file-system organization.

Computer systems may also have varying numbers of file systems, and the file systems may be of varying types. For example, a typical Solaris system may have dozens of file systems of a dozen different types, as shown in the file-system list in Figure 15.2.

In this book, we consider only general-purpose file systems. It is worth noting, though, that there are many special-purpose file systems. Consider the types of file systems in the Solaris example mentioned above:

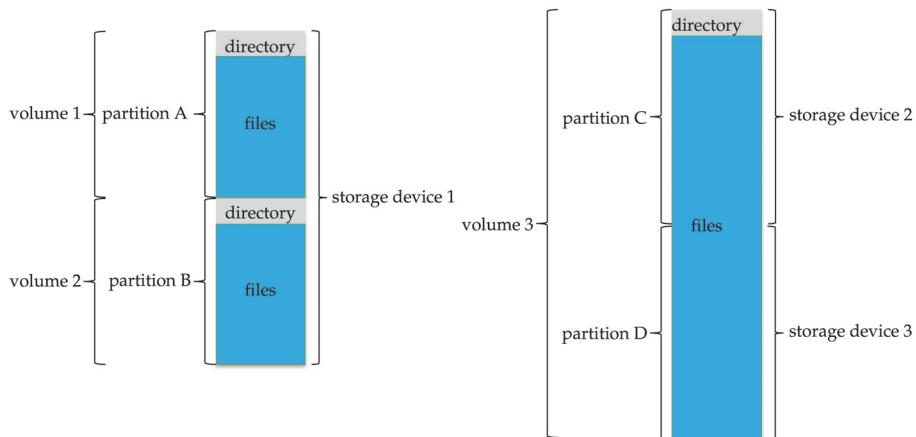


Figure 15.1 A typical storage device organization.

- **tmpfs**—a “temporary” file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **objfs**—a “virtual” file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- **ctfs**—a virtual file system that maintains “contract” information to manage which processes start when the system boots and must continue to run during operation
- **lofs**—a “loop back” file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

The file systems of computers, then, can be extensive. Even within a file system, it is useful to segregate files into groups and manage and act on those groups. This organization involves the use of directories (see Section 14.3).

15.2 File-System Mounting

Just as a file must be opened before it can be used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple file-system-containing volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Some operating systems require that a file-system type be provided, while others inspect the structures of the device

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Figure 15.2 Solaris file systems.

and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as /home; then, to access the directory structure within that file system, we could precede the directory names with /home, as in /home/jane. Mounting that file system under /users would result in the path name /users/jane, which we could use to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.

To illustrate file mounting, consider the file system depicted in Figure 15.3, where the triangles represent subtrees of directories that are of interest. Figure 15.3(a) shows an existing file system, while Figure 15.3(b) shows an unmounted volume residing on /device/dsk. At this point, only the files on the existing file system can be accessed. Figure 15.4 shows the effects of mounting the volume residing on /device/dsk over /users. If the volume is unmounted, the file system is restored to the situation depicted in Figure 15.3.

Systems impose semantics to clarify functionality. For example, a system may disallow a mount over a directory that contains files; or it may make the

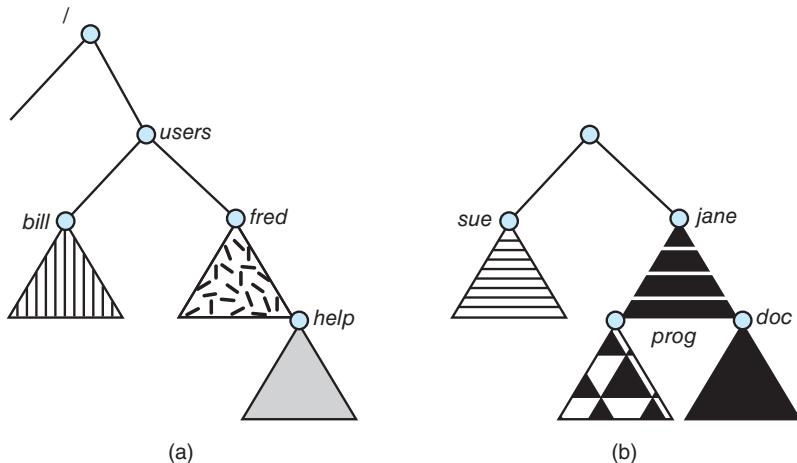


Figure 15.3 File system. (a) Existing system. (b) Unmounted volume.

mounted file system available at that directory and obscure the directory's existing files until the file system is unmounted, terminating the use of the file system and allowing access to the original files in that directory. As another example, a system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

Consider the actions of the macOS operating system. Whenever the system encounters a disk for the first time (either at boot time or while the system is running), the macOS operating system searches for a file system on the device. If it finds one, it automatically mounts the file system under the `/Volumes` directory, adding a folder icon labeled with the name of the file system (as stored in the device directory). The user is then able to click on the icon and thus display the newly mounted file system.

The Microsoft Windows family of operating systems maintains an extended two-level directory structure, with devices and volumes assigned drive letters. Each volume has a general graph directory structure associated

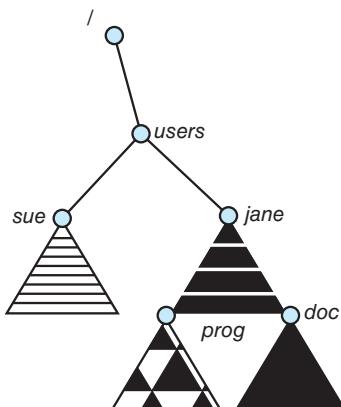


Figure 15.4 Volume mounted at /users.

with its drive letter. The path to a specific file takes the form `drive-letter:\path\to\file`. The more recent versions of Windows allow a file system to be mounted anywhere in the directory tree, just as UNIX does. Windows operating systems automatically discover all devices and mount all located file systems at boot time. In some systems, like UNIX, the mount commands are explicit. A system configuration file contains a list of devices and mount points for automatic mounting at boot time, but other mounts may be executed manually.

Issues concerning file system mounting are further discussed in Section 15.3 and in Section C.7.5.

15.3 Partitions and Mounting

The layout of a disk can have many variations, depending on the operating system and volume management software. A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks. The former layout is discussed here, while the latter, which is more appropriately considered a form of RAID, is covered in Section 11.8.

Each partition can be either “raw,” containing no file system, or “cooked,” containing a file system. **Raw disk** is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system. Likewise, some databases use raw disk and format the data to suit their needs. Raw disk can also hold information needed by disk RAID systems, such as bitmaps indicating which blocks are mirrored and which have changed and need to be mirrored. Similarly, raw disk can contain a miniature database holding RAID configuration information, such as which disks are members of each RAID set. Raw disk use is discussed in Section 11.5.1.

If a partition contains a file system that is bootable—that has a properly installed and configured operating system—then the partition also needs boot information, as described in Section 11.5.2. This information has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format. Rather, boot information is usually a sequential series of blocks loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte. This image, the **bootstrap loader**, in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing.

The boot loader can contain more than the instructions for booting a specific operating system. For instance, many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system. How does the system know which one to boot? A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space. Once loaded, it can boot one of the operating systems available on the drive. The drive can have multiple partitions, each containing a different type of file system and a different operating system. Note that if the boot loader does not understand a particular file-system format, an operating system stored on that file system is not bootable. This is one of the reasons only some file systems are supported as root file systems for any given operating system.

The **root partition** selected by the boot loader, which contains the operating-system kernel and sometimes other system files, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system. As part of a successful mount operation, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention. Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system. The details of this function depend on the operating system.

Microsoft Windows-based systems mount each volume in a separate name space, denoted by a letter and a colon, as mentioned earlier. To record that a file system is mounted at F:, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to F:. When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory. Later versions of Windows can mount a file system at any point within the existing directory structure.

On UNIX, file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there. The mount table entry contains a pointer to the superblock of the file system on that device. This scheme enables the operating system to traverse its directory structure, switching seamlessly among file systems of varying types.

15.4 File Sharing

The ability to share files is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user-oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

In this section, we examine more aspects of file sharing. We begin by discussing general issues that arise when multiple users share files. Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems; we discuss that challenge as well. Finally, we consider what to do about conflicting actions occurring on shared files. For instance, if multiple users are writing to a file, should all the writes be allowed to occur, or should the operating system protect the users' actions from one another?

15.4.1 Multiple Users

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users

by default or require that a user specifically grant access to the files. These are the issues of access control and protection, which are covered in Section 13.4.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system. Although many approaches have been taken to meet this requirement, most systems have evolved to use the concepts of file (or directory) **owner** (or **user**) and **group**. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file. For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner.

The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Many systems have multiple local file systems, including volumes of a single disk or multiple volumes on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted. But consider an external disk that can be moved between systems. What if the IDs on the systems are different? Care must be taken to be sure that IDs match between systems when devices move between them or that file ownership is reset when such a move occurs. (For example, we can create a new user ID and set all files on the portable disk to that ID, to be sure no files are accidentally accessible to existing users.)

15.5 Virtual File Systems

As we've seen, modern operating systems must concurrently support multiple types of file systems. But how does an operating system allow multiple types of file systems to be integrated into a directory structure? And how can users seamlessly move between file-system types as they navigate the file-system space? We now discuss some of these implementation details.

An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type. Instead, however, most operating systems, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation. The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS. Users can access files contained within multiple file systems on the local drive or even on file systems available across the network.

Data structures and procedures are used to isolate the basic system-call functionality from the implementation details. Thus, the file-system implementation consists of three major layers, as depicted schematically in Figure 15.5. The first layer is the file-system interface, based on the `open()`, `read()`, `write()`, and `close()` calls and on file descriptors.

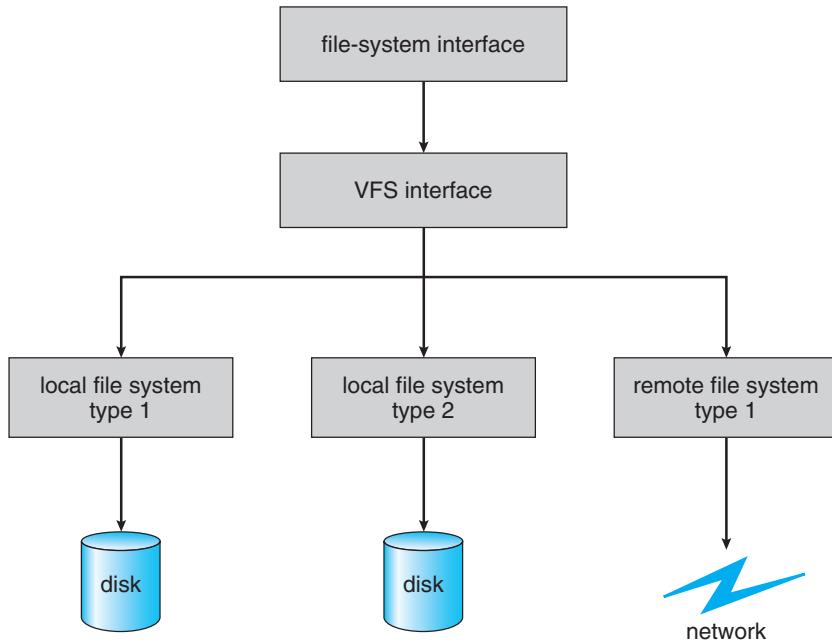


Figure 15.5 Schematic view of a virtual file system.

The second layer is called the **virtual file system (VFS)** layer. The VFS layer serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode**, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node (file or directory).

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

The VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the NFS protocol procedures (or other protocol procedures for other network file systems) for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.

Let's briefly examine the VFS architecture in Linux. The four main object types defined by the Linux VFS are:

- The **inode object**, which represents an individual file
- The **fil object**, which represents an open file
- The **superblock object**, which represents an entire file system
- The **dentry object**, which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that may be implemented. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that particular object. For example, an abbreviated API for some of the operations for the file object includes:

- `int open(. . .)`—Open a file.
- `int close(. . .)`—Close an already-open file.
- `ssize_t read(. . .)`—Read from a file.
- `ssize_t write(. . .)`—Write to a file.
- `int mmap(. . .)`—Memory-map a file.

An implementation of the file object for a specific file type is required to implement each function specified in the definition of the file object. (The complete definition of the file object is specified in the file `struct file_operations`, which is located in the file `/usr/include/linux/fs.h`.)

Thus, the VFS software layer can perform an operation on one of these objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a disk file, a directory file, or a remote file. The appropriate function for that file's `read()` operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

15.6 Remote File Systems

With the advent of networks (Chapter 19), communication among remote computers became possible. Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.

Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like `ftp`. The second major method uses a **distributed fil system (DFS)**, in which remote directories are visible from a local machine. In some ways, the third method, the **World Wide Web**, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for `ftp`) are used to transfer files. Increasingly, cloud computing (Section 1.10.5) is being used for file sharing as well.

ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system. The World Wide Web uses anonymous file exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity, as we describe in this section.

15.6.1 The Client-Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the **server**, and the machine seeking access to the files is the **client**. The client-server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.

The server usually specifies the available files on a volume or directory level. Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be **spoofed**, or imitated. As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys. Unfortunately, with security come many challenges, including ensuring compatibility of the client and server (they must use the same encryption algorithms) and security of key exchanges (intercepted keys could again allow unauthorized access). Because of the difficulty of solving these problems, unsecure authentication methods are most commonly used.

In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information, by default. In this scheme, the user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files. Consider the example of a user who has an ID of 1000 on the client and 2000 on the server. A request from the client to the server for a specific file will not be handled appropriately, as the server will determine if user 1000 has access to the file rather than basing the determination on the real user ID of 2000. Access is thus granted or denied based on incorrect authentication information. The server must trust the client to present the correct user ID. Note that the NFS protocols allow many-to-many relationships. That is, many servers can provide files to many clients. In fact, a given machine can be both a server to some NFS clients and a client of other NFS servers.

Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol. Typically, a file-open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested. The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then can perform read, write, and other operations on the file. The client closes the file when access is completed. The operating system may apply semantics similar to those for a local file-system mount or may use different semantics.

15.6.2 Distributed Information Systems

To make client–server systems easier to manage, **distributed information systems**, also known as **distributed naming services**, provide unified access to the information needed for remote computing. The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet. Before DNS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts. Obviously, this methodology was not scalable! DNS is further discussed in Section 19.3.1.

Other distributed information systems provide *user name/password/user ID/group ID* space for a distributed facility. UNIX systems have employed a wide variety of distributed information methods. Sun Microsystems (now part of Oracle Corporation) introduced **yellow pages** (since renamed **network information service**, or **NIS**), and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like. Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted (in clear text) and identifying hosts by IP address. Sun’s NIS+ was a much more secure replacement for NIS but was much more complicated and was not widely adopted.

In the case of Microsoft’s **common Internet file system (CIFS)**, network information is used in conjunction with user authentication (user name and password) to create a network login that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match from machine to machine (as with NFS). Microsoft uses **active directory** as a distributed naming structure to provide a single name space for users. Once established, the distributed naming facility is used by all clients and servers to authenticate users via Microsoft’s version of the **Kerberos** network authentication protocol (<https://web.mit.edu/kerberos/>).

The industry is moving toward use of the **lightweight directory-access protocol (LDAP)** as a secure distributed naming mechanism. In fact, active directory is based on LDAP. Oracle Solaris and most other major operating systems include LDAP and allow it to be employed for user authentication as well as system-wide retrieval of information, such as availability of printers. Conceivably, one distributed LDAP directory could be used by an organization to store all user and resource information for all the organization’s computers. The result would be secure single sign-on for users, who would enter their authentication information once for access to all computers within the organization. It would also ease system-administration efforts by combining, in one location, information that is currently scattered in various files on each system or in different distributed information services.

15.6.3 Failure Modes

Local file systems can fail for a variety of reasons, including failure of the drive containing the file system, corruption of the directory structure or other disk-management information (collectively called **metadata**), disk-controller failure, cable failure, and host-adapter failure. User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention may be required to repair the damage.

Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues. Although some networks have built-in resiliency, including multiple paths between hosts, many do not. Any single failure can thus interrupt the flow of DFS commands.

Consider a client in the midst of using a remote file system. It has files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable. This scenario is rather common, so it would not be appropriate for the client system to act as it would if a local file system were lost. Rather, the system can either terminate all operations to the lost server or delay operations until the server is again reachable. These failure semantics are defined and implemented as part of the remote-file-system protocol. Termination of all operations can result in users' losing data—and patience. Thus, most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again.

To implement this kind of recovery from failure, some kind of **state information** may be maintained on both the client and the server. If both server and client maintain knowledge of their current activities and open files, then they can seamlessly recover from a failure. In the situation where the server crashes but must recognize that it has remotely mounted exported file systems and opened files, NFS Version 3 takes a simple approach, implementing a **stateless** DFS. In essence, it assumes that a client request for a file read or write would not have occurred unless the file system had been remotely mounted and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation. Similarly, it does not track which clients have the exported volumes mounted, again assuming that if a request comes in, it must be legitimate. While this stateless approach makes NFS resilient and rather easy to implement, it also makes it unsecure. For example, forged read or write requests could be allowed by an NFS server. These issues are addressed in the industry standard NFS Version 4, in which NFS is made stateful to improve its security, performance, and functionality.

15.7 Consistency Semantics

Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. In particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

Consistency semantics are directly related to the process synchronization algorithms of Chapter 6. However, the complex algorithms of that chapter tend

not to be implemented in the case of file I/O because of the great latencies and slow transfer rates of disks and networks. For example, performing an atomic transaction to a remote disk could involve several network communications, several disk reads and writes, or both. Systems that attempt such a full set of functionalities tend to perform poorly. A successful implementation of complex sharing semantics can be found in the Andrew file system.

For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the `open()` and `close()` operations. The series of accesses between the `open()` and `close()` operations makes up a **file session**. To illustrate the concept, we sketch several prominent examples of consistency semantics.

15.7.1 UNIX Semantics

The UNIX file system (Chapter 19) uses the following consistency semantics:

- Writes to an open file by a user are visible immediately to other users who have this file open.
- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.

15.7.2 Session Semantics

The Andrew file system (OpenAFS) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay. Almost no constraints are enforced on scheduling accesses.

15.7.3 Immutable-Shared-Files Semantics

A unique approach is that of **immutable shared files**. Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties: its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed. The implementation of these semantics in a distributed system (Chapter 19) is simple, because the sharing is disciplined (read-only).

15.8 NFS

Network file systems are commonplace. They are typically integrated with the overall directory structure and interface of the client system. NFS is a good example of a widely used, well implemented client–server network file system. Here, we use it as an example to explore the implementation details of network file systems.

NFS is both an implementation and a specification of a software system for accessing remote files across LANs (or even WANs). NFS is part of ONC+, which most UNIX vendors and some PC operating systems support. The implementation described here is part of the Solaris operating system, which is a modified version of UNIX SVR4. It uses either the TCP or UDP/IP protocol (depending on the interconnecting network). The specification and the implementation are intertwined in our description of NFS. Whenever detail is needed, we refer to the Solaris implementation; whenever the description is general, it applies to the specification also.

There are multiple versions of NFS, with the latest being Version 4. Here, we describe Version 3, which is the version most commonly deployed.

15.8.1 Overview

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems (on explicit request) in a transparent manner. Sharing is based on a client–server relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines. To ensure machine independence, sharing of a remote file system affects only the client machine and no other machine.

So that a remote directory will be accessible in a transparent manner from a particular machine—say, from *M1*—a client of that machine must first carry out a mount operation. The semantics of the operation involve mounting a remote directory over a directory of a local file system. Once the mount operation is completed, the mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory. The local directory becomes the name of the root of the newly mounted directory. Specification of the remote directory as an argument for the mount operation is not done transparently; the location (or host name) of the remote directory has to be provided. However, from then on, users on machine *M1* can access files in the remote directory in a totally transparent manner.

To illustrate file mounting, consider the file system depicted in Figure 15.6, where the triangles represent subtrees of directories that are of interest. The figure shows three independent file systems of machines named *U*, *S1*, and *S2*. At this point, on each machine, only the local files can be accessed. Figure 15.7(a) shows the effects of mounting *S1:/usr/shared* over *U:/usr/local*. This figure depicts the view users on *U* have of their file system. After the mount is complete, they can access any file within the *dir1* directory using the prefix */usr/local/dir1*. The original directory */usr/local* on that machine is no longer visible.

Subject to access-rights accreditation, any file system, or any directory within a file system, can be mounted remotely on top of any local directory.

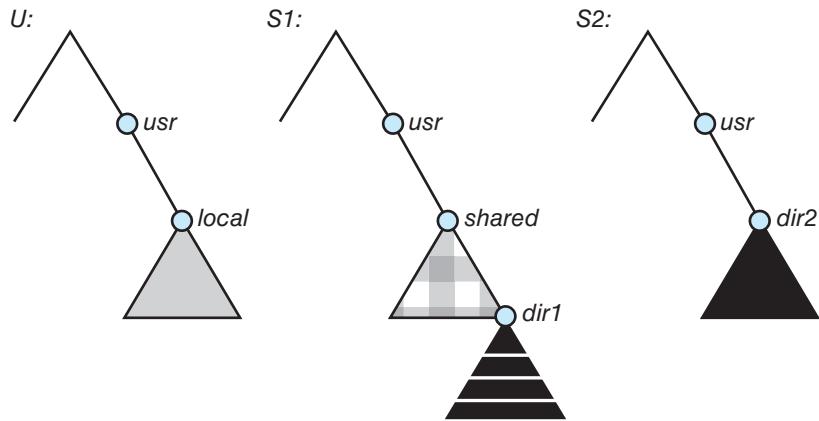


Figure 15.6 Three independent file systems.

Diskless workstations can even mount their own roots from servers. Cascading mounts are also permitted in some NFS implementations. That is, a file system can be mounted over another file system that is remotely mounted, not local. A machine is affected by only those mounts that it has itself invoked. Mounting a remote file system does not give the client access to other file systems that were, by chance, mounted over the former file system. Thus, the mount mechanism does not exhibit a transitivity property.

In Figure 15.7(b), we illustrate cascading mounts. The figure shows the result of mounting S2:/usr/dir2 over U:/usr/local/dir1, which is already remotely mounted from S1. Users can access files within dir2 on *U* using the prefix /usr/local/dir1. If a shared file system is mounted over a user's home directories on all machines in a network, the user can log into any workstation and get his or her home environment. This property permits user mobility.

One of the design goals of NFS was to operate in a heterogeneous environment of different machines, operating systems, and network architectures. The

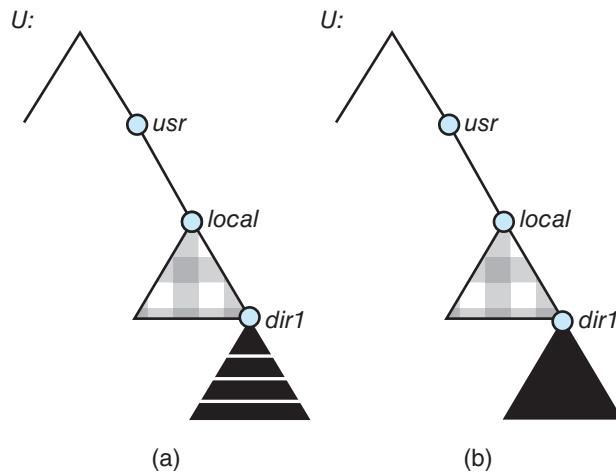


Figure 15.7 Mounting in NFS. (a) Mounts. (b) Cascading mounts.

NFS specification is independent of these media. This independence is achieved through the use of RPC primitives built on top of an external data representation (XDR) protocol used between two implementation-independent interfaces. Hence, if the system's heterogeneous machines and file systems are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services. Accordingly, two separate protocols are specified for these services: a mount protocol and a protocol for remote file accesses, the **NFS protocol**. The protocols are specified as sets of RPCs. These RPCs are the building blocks used to implement transparent remote file access.

15.8.2 The Mount Protocol

The **mount protocol** establishes the initial logical connection between a server and a client. In Solaris, each machine has a server process, outside the kernel, performing the protocol functions.

A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The mount request is mapped to the corresponding RPC and is forwarded to the mount server running on the specific server machine. The server maintains an **export list** that specifies local file systems that it exports for mounting, along with names of machines that are permitted to mount them. (In Solaris, this list is the `/etc/dfs/dfstab`, which can be edited only by a superuser.) The specification can also include access rights, such as read only. To simplify the maintenance of export lists and mount tables, a distributed naming scheme can be used to hold this information and make it available to appropriate clients.

Recall that any directory within an exported file system can be mounted remotely by an accredited machine. A component unit is such a directory. When the server receives a mount request that conforms to its export list, it returns to the client a file handle that serves as the key for further accesses to files within the mounted file system. The file handle contains all the information that the server needs to distinguish an individual file it stores. In UNIX terms, the file handle consists of a file-system identifier and an inode number to identify the exact mounted directory within the exported file system.

The server also maintains a list of the client machines and the corresponding currently mounted directories. This list is used mainly for administrative purposes—for instance, for notifying all clients that the server is going down. Only through addition and deletion of entries in this list can the server state be affected by the mount protocol.

Usually, a system has a static mounting preconfiguration that is established at boot time (`/etc/vfstab` in Solaris); however, this layout can be modified. In addition to the actual mount procedure, the mount protocol includes several other procedures, such as unmount and return export list.

15.8.3 The NFS Protocol

The NFS protocol provides a set of RPCs for remote file operations. The procedures support the following operations:

- Searching for a file within a directory
- Reading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

These procedures can be invoked only after a file handle for the remotely mounted directory has been established.

The omission of open and close operations is intentional. A prominent feature of NFS servers is that they are stateless. Servers do not maintain information about their clients from one access to another. No parallels to UNIX's open-files table or file structures exist on the server side. Consequently, each request has to provide a full set of arguments, including a unique file identifier and an absolute offset inside the file for the appropriate operations. The resulting design is robust; no special measures need be taken to recover a server after a crash. File operations must be idempotent for this purpose—that is, the same operation performed multiple times must have the same effect as if it had only been performed once. To achieve idempotence, every NFS request has a sequence number, allowing the server to determine if a request has been duplicated or if any are missing.

Maintaining the list of clients that we mentioned seems to violate the statelessness of the server. However, this list is not essential for the correct operation of the client or the server, and hence it does not need to be restored after a server crash. Consequently, it may include inconsistent data and is treated as only a hint.

A further implication of the stateless-server philosophy and a result of the synchrony of an RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before results are returned to the client. That is, a client can cache write blocks, but when it flushes them to the server, it assumes that they have reached the server's disks. The server must write all NFS data synchronously. Thus, a server crash and recovery will be invisible to a client; all blocks that the server is managing for the client will be intact. The resulting performance penalty can be large, because the advantages of caching are lost. Performance can be increased by using storage with its own nonvolatile cache (usually battery-backed-up memory). The disk controller acknowledges the disk write when the write is stored in the nonvolatile cache. In essence, the host sees a very fast synchronous write. These blocks remain intact even after a system crash and are written from this stable storage to disk periodically.

A single NFS write procedure call is guaranteed to be atomic and is not intermixed with other write calls to the same file. The NFS protocol, however, does not provide concurrency-control mechanisms. A `write()` system call may be broken down into several RPC writes, because each NFS write or read call can contain up to 8 KB of data and UDP packets are limited to 1,500 bytes. As a result, two users writing to the same remote file may get their data intermixed. The claim is that, because lock management is inherently stateful, a service outside the NFS should provide locking (and Solaris does). Users are advised to coordinate access to shared files using mechanisms outside the scope of NFS.

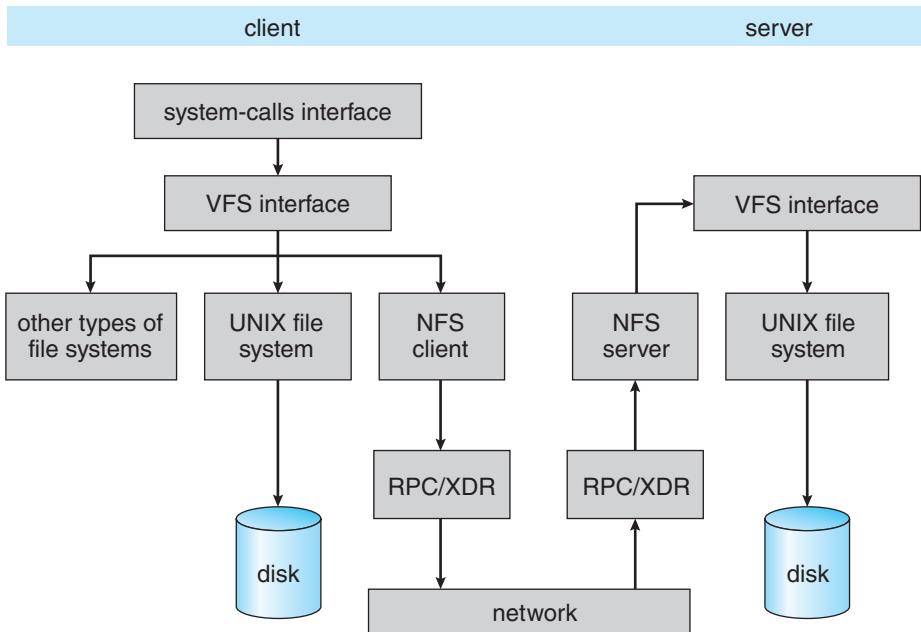


Figure 15.8 Schematic view of the NFS architecture.

NFS is integrated into the operating system via a VFS. As an illustration of the architecture, let's trace how an operation on an already-open remote file is handled (follow the example in Figure 15.8). The client initiates the operation with a regular system call. The operating-system layer maps this call to a VFS operation on the appropriate vnode. The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the NFS service layer at the remote server. This call is reinjected to the VFS layer on the remote system, which finds that it is local and invokes the appropriate file-system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, a machine may be a client, or a server, or both. The actual service on each server is performed by kernel threads.

15.8.4 Path-Name Translation

Path-name translation in NFS involves the parsing of a path name such as `/usr/local/dir1/file.txt` into separate directory entries, or components: (1) `usr`, (2) `local`, and (3) `dir1`. Path-name translation is done by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode. Once a mount point is crossed, every component lookup causes a separate RPC to the server. This expensive path-name-traversal scheme is needed, since the layout of each client's logical name space is unique, dictated by the mounts the client has performed. It would be much more efficient to hand a server a path name and receive a target vnode once a mount point is encountered. At any point, however, there might be another mount point for the particular client of which the stateless server is unaware.

So that lookup is fast, a directory-name-lookup cache on the client side holds the vnodes for remote directory names. This cache speeds up references to files with the same initial path name. The directory cache is discarded when attributes returned from the server do not match the attributes of the cached vnode.

Recall that some implementations of NFS allow mounting a remote file system on top of another already-mounted remote file system (a cascading mount). When a client has a cascading mount, more than one server can be involved in a path-name traversal. However, when a client does a lookup on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory.

15.8.5 Remote Operations

With the exception of opening and closing files, there is an almost one-to-one correspondence between the regular UNIX system calls for file operations and the NFS protocol RPCs. Thus, a remote file operation can be translated directly to the corresponding RPC. Conceptually, NFS adheres to the remote-service paradigm; but in practice, buffering and caching techniques are employed for the sake of performance. No direct correspondence exists between a remote operation and an RPC. Instead, file blocks and file attributes are fetched by the RPCs and are cached locally. Future remote operations use the cached data, subject to consistency constraints.

There are two caches: the file-attribute (inode-information) cache and the file-blocks cache. When a file is opened, the kernel checks with the remote server to determine whether to fetch or revalidate the cached attributes. The cached file blocks are used only if the corresponding cached attributes are up to date. The attribute cache is updated whenever new attributes arrive from the server. Cached attributes are, by default, discarded after 60 seconds. Both read-ahead and delayed-write techniques are used between the server and the client. Clients do not free delayed-write blocks until the server confirms that the data have been written to disk. Delayed-write is retained even when a file is opened concurrently, in conflicting modes. Hence, UNIX semantics (Section 15.7.1) are not preserved.

Tuning the system for performance makes it difficult to characterize the consistency semantics of NFS. New files created on a machine may not be visible elsewhere for 30 seconds. Furthermore, writes to a file at one site may or may not be visible at other sites that have this file open for reading. New opens of a file observe only the changes that have already been flushed to the server. Thus, NFS provides neither strict emulation of UNIX semantics nor the session semantics of Andrew (Section 15.7.2). In spite of these drawbacks, the utility and good performance of the mechanism make it the most widely used multi-vendor-distributed system in operation.

15.9 Summary

- General-purpose operating systems provide many file-system types, from special-purpose through general.

- Volumes containing file systems can be mounted into the computer's file-system space.
- Depending on the operating system, the file-system space is seamless (mounted file systems integrated into the directory structure) or distinct (each mounted file system having its own designation).
- At least one file system must be bootable for the system to be able to start —that is, it must contain an operating system. The boot loader is run first; it is a simple program that is able to find the kernel in the file system, load it, and start its execution. Systems can contain multiple bootable partitions, letting the administrator choose which to run at boot time.
- Most systems are multi-user and thus must provide a method for file sharing and file protection. Frequently, files and directories include metadata, such as owner, user, and group access permissions.
- Mass storage partitions are used either for raw block I/O or for file systems. Each file system resides in a volume, which can be composed of one partition or multiple partitions working together via a volume manager.
- To simplify implementation of multiple file systems, an operating system can use a layered approach, with a virtual file-system interface making access to possibly dissimilar file systems seamless.
- Remote file systems can be implemented simply by using a program such as `ftp` or the web servers and clients in the World Wide Web, or with more functionality via a client–server model. Mount requests and user IDs must be authenticated to prevent unapproved access.
- Client–server facilities do not natively share information, but a distributed information system such as DNS can be used to allow such sharing, providing a unified user name space, password management, and system identification. For example, Microsoft CIFS uses active directory, which employs a version of the Kerberos network authentication protocol to provide a full set of naming and authentication services among the computers in a network.
- Once file sharing is possible, a consistency semantics model must be chosen and implemented to moderate multiple concurrent access to the same file. Semantics models include UNIX, session, and immutable-shared-files semantics.
- NFS is an example of a remote file system, providing clients with seamless access to directories, files, and even entire file systems. A full-featured remote file system includes a communication protocol with remote operations and path-name translation.

Practice Exercises

- 15.1 Explain how the VFS layer allows an operating system to support multiple types of file systems easily.
- 15.2 Why have more than one file system type on a given system?

- 15.3 On a Unix or Linux system that implements the procfs file system, determine how to use the procfs interface to explore the process name space. What aspects of processes can be viewed via this interface? How would the same information be gathered on a system lacking the procfs file system?
- 15.4 Why do some systems integrate mounted file systems into the root file system naming structure, while others use a separate naming method for mounted file systems?
- 15.5 Given a remote file access facility such as ftp, why were remote file systems like NFS created?

Further Reading

The internals of the BSD UNIX system are covered in full in [McKusick et al. (2015)]. Details concerning file systems for Linux can be found in [Love (2010)].

The network file system (NFS) is discussed in [Callaghan (2000)]. NFS Version 4 is a standard described at <http://www.ietf.org/rfc/rfc3530.txt>. [Ousterhout (1991)] discusses the role of distributed state in networked file systems. NFS and the UNIX file system (UFS) are described in [Mauro and McDougall (2007)].

The Kerberos network authentication protocol is explored in <https://web.mit.edu/kerberos/>.

Bibliography

[Callaghan (2000)] B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[Mauro and McDougall (2007)] J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2007).

[McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System—Second Edition*, Pearson (2015).

[Ousterhout (1991)] J. Ousterhout. “The Role of Distributed State”. In *CMU Computer Science: a 25th Anniversary Commemorative*, R. F. Rashid, Ed., Addison-Wesley (1991).

Chapter 15 Exercises

- 15.6 Assume that in a particular augmentation of a remote-file-access protocol, each client maintains a name cache that caches translations from file names to corresponding file handles. What issues should we take into account in implementing the name cache?
- 15.7 Given a mounted file system with write operations underway, and a system crash or power loss, what must be done before the file system is remounted if: (a) The file system is not log-structured? (b) The file system is log-structured?
- 15.8 Why do operating systems mount the root file system automatically at boot time?
- 15.9 Why do operating systems require file systems other than root to be mounted?