Preface

Operating systems are an essential part of any computer system. Similarly, a course on operating systems is an essential part of any computer-science education. This field is undergoing rapid change, as computers are now prevalent in virtually every application, from games for children through the most sophisticated planning tools for governments and multinational firms. Yet the fundamental concepts remain clear, and it is on these that we base this book.

We wrote this book as a text for an introductory course in operating systems at the junior or senior undergraduate level or at the first-year graduate level. We hope that practitioners will also find it useful. It provides a clear description of the concepts that underlie operating systems. As prerequisites, we assume that the reader is familiar with basic data structures, computer organization, and a high-level language, such as C. The hardware topics required for an understanding of operating systems are included in Chapter 1. For code examples, we use predominantly C, with some Java, but the reader can still understand the algorithms without a thorough knowledge of these languages.

Concepts are presented using intuitive descriptions. Important theoretical results are covered, but formal proofs are omitted. The bibliographical notes contain pointers to research papers in which results were first presented and proved, as well as references to material for further reading. In place of proofs, figures and examples are used to suggest why we should expect the result in question to be true.

The fundamental concepts and algorithms covered in the book are often based on those used in existing commercial operating systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular operating system. We present a large number of examples that pertain to the most popular and the most innovative operating systems, including Sun Microsystems' Solaris; Linux; Mach; Microsoft MS-DOS, Windows NT, Windows 2000, and Windows XP; DEC VMS and TOPS-20; IBM OS/2; and Apple Mac OS X.

In this text, when we refer to Windows XP as an example operating system, we are implying both Windows XP and Windows 2000. If a feature exists in Windows XP that is not available in Windows 2000, we will state this explicitly.
If a feature exists in Windows 2000 but not in Windows XP, then we will refer specifically to Windows 2000.

Organization of This Book

The organization of this text reflects our many years of teaching operating systems courses. Consideration was also given to the feedback provided by the reviewers of the text, as well as comments submitted by readers of earlier editions. In addition, the content of the text corresponds to the suggestions from *Computing Curricula 2001* for teaching operating systems, published by the Joint Task Force of the IEEE Computing Society and the Association for Computing Machinery (ACM).

On the supporting web page for this text, we provide several sample syllabi that suggest various approaches for using the text in both introductory and advanced operating systems courses. As a general rule, we encourage readers to progress sequentially through the chapters, as this strategy provides the most thorough study of operating systems. However, by using the sample syllabi, a reader can select a different ordering of chapters (or subsections of chapters).

Content of This Book

The text is organized into eight major parts:

- **Overview.** Chapters 1 and 2 explain what operating systems are, what they do, and how they are designed and constructed. They discuss what the common features of an operating system are, what an operating system does for the user, and what it does for the computer-system operator. The presentation is motivational and explanatory in nature. We have avoided a discussion of how things are done internally in these chapters. Therefore, they are suitable for individual readers or for students in lower-level classes who want to learn what an operating system is without getting into the details of the internal algorithms.

- **Process management.** Chapters 3 through 7 describe the process concept and concurrency as the heart of modern operating systems. A process is the unit of work in a system. Such a system consists of a collection of concurrently executing 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). These chapters cover methods for process scheduling, interprocess communication, process synchronization, and deadlock handling. Also included under this topic is a discussion of threads.

- **Memory management.** Chapters 8 and 9 deal with main memory management during the execution of a process. To improve both the utilization of the CPU and the speed of its response to its users, the computer must keep several processes in memory. There are many different memory-management schemes, reflecting various approaches to memory management, and the effectiveness of a particular algorithm depends on the situation.
• **Chapter 19, Real-Time Systems**, is a new chapter focusing on real-time and embedded computing systems, which have requirements different from those of many traditional systems. The chapter provides an overview of real-time computer systems and describes how operating systems must be constructed to meet the stringent timing deadlines of these systems.

• **Chapter 20, Multimedia Systems**, is a new chapter detailing developments in the relatively new area of multimedia systems. Multimedia data differ from conventional data in that multimedia data—such as frames of video—must be delivered (streamed) according to certain time restrictions. The chapter explores how these requirements affect the design of operating systems.

• **Chapter 21, The Linux System**, is the old Chapter 20, updated to reflect changes in the 2.6 kernel—the most recent kernel at the time this text was written.

• **Chapter 22, XP**, has been updated.

• **Chapter 22, Influential Operating Systems**, has been updated.

The old Chapter 21 (Windows 2000) has been moved into Appendix C. As in the previous edition, the appendices are provided online.

**Programming Exercises and Projects**

To emphasize the concepts presented in the text, we have added several programming exercises and projects that use the POSIX and Win32 APIs as well as Java. We have added over 15 new programming exercises that emphasize processes, threads, shared memory, process synchronization, and networking. In addition, we have added several programming projects which are more involved than standard programming exercises. These projects include adding a system call to the Linux kernel, creating a UNIX shell using the `fork()` system call, a multithreaded matrix application, and the producer-consumer problem using shared memory.

**Teaching Supplements and Web Page**

The web page for the book contains such material as a set of slides to accompany the book, model course syllabi, all C and Java source code, and up-to-date errata. The web page also contains the book’s three case-study appendices and the Distributed Communication appendix. The URL is:

http://www.os-book.com

New to this edition is a print supplement called the Student Solutions Manual. Included are problems and exercises with solutions not found in the text that should help students master the concepts presented. You can purchase a print copy of this supplement at Wiley’s website by going to http://www.wiley.com/college/silberschatz and choosing the Student Solutions Manual link.
To obtain restricted supplements, such as the solution guide to the exercises in the text, contact your local John Wiley & Sons sales representative. Note that these supplements are available only to faculty who use this text. You can find your representative at the "Find a Rep?" web page: http://www.jsw-edcv.wiley.com/college/findarep.

Mailing List

We have switched to the mailman system for communication among the users of Operating System Concepts. If you wish to use this facility, please visit the following URL and follow the instructions there to subscribe:

http://mailman.cs.yale.edu/mailman/listinfo/os-book-list

The mailman mailing-list system provides many benefits, such as an archive of postings, as well as several subscription options, including digest and Web only. To send messages to the list, send e-mail to:

os-book-list@cs.yale.edu

Depending on the message, we will either reply to you personally or forward the message to everyone on the mailing list. The list is moderated, so you will receive no inappropriate mail.

Students who are using this book as a text for class should not use the list to ask for answers to the exercises; they will not be provided.

Suggestions

We have attempted to clean up every error in this new edition, but—as happens with operating systems—a few obscure bugs may remain. We would appreciate hearing from you about any textual errors or omissions that you identify.

If you would like to suggest improvements or to contribute exercises, we would also be glad to hear from you. Please send correspondence to os-book@cs.yale.edu.

Acknowledgments

This book is derived from the previous editions, the first three of which were coauthored by James Peterson. Others who helped us with previous editions include Hamid Arabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Campbell, P. C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Racsit Eskicioglu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, Max Hailperin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Bruce Hillyer, Mark Holliday, Ahmed Kamel, Richard Kieburz, Carol Kroll, Morty Kwestel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller,
## Contents

PART ONE • OVERVIEW

Chapter 1 Introduction

1.1 What Operating Systems Do 3  
1.2 Computer-System Organization 6  
1.3 Computer-System Architecture 12  
1.4 Operating-System Structure 15  
1.5 Operating-System Operations 17  
1.6 Process Management 20  
1.7 Memory Management 22  
1.8 Storage Management 22  
1.9 Protection and Security 26  
1.10 Distributed Systems 28  
1.11 Special-Purpose Systems 29  
1.12 Computing Environments 31  
1.13 Summary 34  
1.14 Exercises 36  
Bibliographical Notes 38

Chapter 2 Operating-System Structures

2.1 Operating-System Services 39  
2.2 User Operating-System Interface 41  
2.3 System Calls 43  
2.4 Types of System Calls 47  
2.5 System Programs 55  
2.6 Operating-System Design and Implementation 56  
2.7 Operating-System Structure 58  
2.8 Virtual Machines 64  
2.9 Operating-System Generation 70  
2.10 System Boot 71  
2.11 Summary 72  
Bibliographical Notes 78

PART TWO • PROCESS MANAGEMENT

Chapter 3 Processes

3.1 Process Concept 81  
3.2 Process Scheduling 85  
3.3 Operations on Processes 90  
3.4 Interprocess Communication 96  
3.5 Examples of IPC Systems 102  
3.6 Communication in Client-Server Systems 108  
3.7 Summary 115  
Bibliographical Notes 125
## Chapter 4 Threads

4.1 Overview 127
4.2 Multithreading Models 129
4.3 Thread Libraries 131
4.4 Threading Issues 138
4.5 Operating-System Examples 143
4.6 Summary 146
Exercises 146
Bibliographical Notes 151

## Chapter 5 CPU Scheduling

5.1 Basic Concepts 153
5.2 Scheduling Criteria 157
5.3 Scheduling Algorithms 158
5.4 Multiple-Processor Scheduling 169
5.5 Thread Scheduling 172
5.6 Operating System Examples 173
5.7 Algorithm Evaluation 181
5.8 Summary 185
Exercises 186
Bibliographical Notes 189

## Chapter 6 Process Synchronization

6.1 Background 191
6.2 The Critical-Section Problem 193
6.3 Peterson’s Solution 195
6.4 Synchronization Hardware 197
6.5 Semaphore 200
6.6 Classic Problems of Synchronization 204
6.7 Monitors 209
6.8 Synchronization Examples 217
6.9 Atomic Transactions 222
Exercises 231
Bibliographical Notes 242

## Chapter 7 Deadlocks

7.1 System Model 245
7.2 Deadlock Characterization 247
7.3 Methods for Handling Deadlocks 252
7.4 Deadlock Prevention 253
7.5 Deadlock Avoidance 256
7.6 Deadlock Detection 262
7.7 Recovery From Deadlock 266
7.8 Summary 267
Exercises 268
Bibliographical Notes 271

## PART THREE MEMORY MANAGEMENT

### Chapter 8 Main Memory

8.1 Background 275
8.2 Swapping 282
8.3 Contiguous Memory Allocation 284
8.4 Paging 288
8.5 Structure of the Page Table 297
8.6 Segmentation 302
8.7 Example: The Intel Pentium 305
8.8 Summary 309
Exercises 310
Bibliographical Notes 312
free-space list, 429
free-space management (disks), 429-431
  bit vector, 429-430
  counting, 431
  grouping, 431
  linked list, 430-431
front-end processors, 523
FTP, see file transfer protocol
ftp, 398
full backup, 436
fully distributed deadlock-detection algorithm, 681-683

G
Gantt chart, 159
garbage collection, 68, 395
gateways, 626
GB (gigabyte), 6
gcc (GNU C compiler), 740
GDT (global descriptor table), 306
general graph directories, 394-395
gigabyte (GB), 6
global descriptor table (GDT), 306
global ordering, 664-666
  hold-and-wait condition (deadlocks), 253-254
  hashing techniques, 100
  hash value (message digest), 582
  heaps, 832, 835-836
  heavyweight processes, 127
  hierarchical paging, 297-300
  hierarchical storage management (HSM), 483
  high availability, 14
  high performance, 786
  hijacking, session, 561
  hit ratio, 294, 358
  hive, 810
  hold-and-wait condition (deadlocks), 253-254
  holes, 286
  holographic storage, 480
  homogeneity, 169
  host adapter, 496
  host-attached storage, 455
  host controller, 453
  hot spare disks, 475
  hot-standby mode, 15
  HSM (hierarchical storage management), 483
  human security, 562
  Hydra, 547-549
  hyperspace, 797
  hyperthreading technology, 171

hands-on computer systems, set
  interactive computer systems
happened-before relation, 664-666
hard affinity, 170
hard-coding techniques, 100
hard errors, 465
hard links, 394
hard real-time systems, 696, 722
hardware, 4
  I/O systems, 496-505
    direct memory access, 503-504
    interrupts, 499-503
    polling, 498-499
  for storing page tables, 292-294
  synchronization, 197-200
hardware-abstraction layer (HAL), 778
  hardware objects, 593
  hashed page tables, 300
  hash functions, 582
  hash tables, 420
  hash value (message digest), 582
  heaps, 832, 835-836
  heavyweight processes, 127
  hierarchical paging, 297-300
  hierarchical storage management (HSM), 483
  high availability, 14
  high performance, 786
  hijacking, session, 561
  hit ratio, 294, 358
  hive, 810
  hold-and-wait condition (deadlocks), 253-254
  holes, 286
  holographic storage, 480
  homogeneity, 169
  host adapter, 496
  host-attached storage, 455
  host controller, 453
  hot spare disks, 475
  hot-standby mode, 15
  HSM (hierarchical storage management), 483
  human security, 562
  Hydra, 547-549
  hyperspace, 797
  hyperthreading technology, 171

G
Gantt chart, 159
garbage collection, 68, 395
gateways, 626
GB (gigabyte), 6
gcc (GNU C compiler), 740
GDT (global descriptor table), 306
general graph directories, 394-395
gigabyte (GB), 6
global descriptor table (GDT), 306
global ordering, 664-666
  hold-and-wait condition (deadlocks), 253-254
  hashing techniques, 100
  hash value (message digest), 582
  heaps, 832, 835-836
  heavyweight processes, 127
  hierarchical paging, 297-300
  hierarchical storage management (HSM), 483
  high availability, 14
  high performance, 786
  hijacking, session, 561
  hit ratio, 294, 358
  hive, 810
  hold-and-wait condition (deadlocks), 253-254
  holes, 286
  holographic storage, 480
  homogeneity, 169
  host adapter, 496
  host-attached storage, 455
  host controller, 453
  hot spare disks, 475
  hot-standby mode, 15
  HSM (hierarchical storage management), 483
  human security, 562
  Hydra, 547-549
  hyperspace, 797
  hyperthreading technology, 171

hands-on computer systems, set
  interactive computer systems
happened-before relation, 664-666
hard affinity, 170
hard-coding techniques, 100
hard errors, 465
hard links, 394
hard real-time systems, 696, 722
hardware, 4
  I/O systems, 496-505
    direct memory access, 503-504
    interrupts, 499-503
    polling, 498-499
  for storing page tables, 292-294
  synchronization, 197-200
hardware-abstraction layer (HAL), 778
  hardware objects, 593
  hashed page tables, 300
  hash functions, 582
  hash tables, 420
  hash value (message digest), 582
  heaps, 832, 835-836
  heavyweight processes, 127
  hierarchical paging, 297-300
  hierarchical storage management (HSM), 483
  high availability, 14
  high performance, 786
  hijacking, session, 561
  hit ratio, 294, 358
  hive, 810
  hold-and-wait condition (deadlocks), 253-254
  holes, 286
  holographic storage, 480
  homogeneity, 169
  host adapter, 496
  host-attached storage, 455
  host controller, 453
  hot spare disks, 475
  hot-standby mode, 15
  HSM (hierarchical storage management), 483
  human security, 562
  Hydra, 547-549
  hyperspace, 797
  hyperthreading technology, 171
In other cases, a user sits at a terminal connected to a **mainframe** or **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization—to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

In still other cases, users sit at **workstations** connected to networks of other workstations and **servers**. These users have dedicated resources at their disposal, but they also share resources such as networking and servers—file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Recently, many varieties of handheld computers have come into fashion. Most of these devices are standalone units for individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems and networking. Because of power, speed, and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per amount of battery life is important as well.

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 are designed primarily for automatic control 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, file-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. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

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

We have looked at the operating system’s role from the views of the user and of the system. How, though, 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 user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. 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 1 megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are entirely based on graphical windowing systems. (A kilobyte, or KB, is 1,024 bytes; a megabyte, or MB, is $1,024^2$ bytes; and a gigabyte, or GB, is $1,024^3$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.) A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being systems programs and application programs. This last definition is the one that we generally follow.

The matter of what constitutes an operating system has become increasingly important. 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 the operating system. As a result, it was found guilty of using its operating system monopoly to limit competition.

1.2 Computer-System Organization

Before we can explore the details of how computer systems operate, we need a general knowledge of the structure of a computer system. In this section, we look at several parts of this structure to round out our background knowledge. The section is mostly concerned with computer-system organization, so you can skim or skip it if you already understand the concepts.

1.2.1 Computer-System Operation

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 to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or bootstrap program, tends to be simple. Typically, it is stored in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM), known by the general term firmware, within the computer hardware. 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 to start executing that system. To accomplish this goal, the bootstrap program must locate and load into memory the operating-system kernel. The operating system starts executing the first process, such as "init," and waits for some event to occur.

The occurrence of an event is usually signaled by an interrupt from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a system call (also called a monitor call).

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 time line of this operation is shown in Figure 1.3.

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.

Figure 1.2 A modern computer system.

Figure 1.3 Interrupt time line for a single process doing output.
Chapter 1  Introduction

generally decreases, whereas the access time generally increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and semiconductor memory have become faster and cheaper. The top four levels of memory in Figure 1.4 may be constructed using semiconductor memory.

In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. As mentioned earlier, volatile storage loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to nonvolatile storage for safekeeping. In the hierarchy shown in Figure 1.4, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile. An electronic disk can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If normal power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into the RAM. Another form of electronic disk is flash memory, which is popular in cameras and personal digital assistants (PDAs), in robots, and increasingly as removable storage on general-purpose computers. Flash memory is slower than DRAM but needs no power to retain its contents. Another form of nonvolatile storage is NVRAM, which is DRAM with battery backup power. This memory can be as fast as DRAM but has a limited duration in which it is nonvolatile.

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

1.2.3 I/O Structure

Storage is only one of many types of I/O devices within a computer. 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. Therefore, we now provide an overview of I/O.

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, there may be more than one attached device. For instance, seven or more devices can be attached to the small computer-systems interface (SCSI) controller. 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
Figure 1.5 How a modern computer system works.

driver understands the device controller and presents a uniform interface to the device to the rest of the operating system.

To start an I/O operation, the device driver loads the appropriate registers within 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 via an interrupt that it has finished its operation. The device driver then returns control to 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.

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk 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 its own buffer storage to 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 1.5 shows the interplay of all components of a computer system.
1.12 Computing Environments

1.12.3 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.

- A peer acting as a client must first 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.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enable peers to exchange files with one another. The Napster system uses an approach similar to the first type described above: a centralized server maintains an index of all files stored on peer nodes in the Napster network, and the actual exchanging of files takes place between the peer nodes. The Gnutella system uses a technique similar to the second type: a client broadcasts file requests to other nodes in the system, and nodes that can service the request respond directly to the client. The future of exchanging files remains uncertain because...
commercial versions of UNIX such as Solaris and IBM's AIX system. However, there has been 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 in the public domain.

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. As a very general rule, many UNIX users prefer a command-line interface as they often provide powerful shell interfaces. Alternatively, most Windows users are pleased to use the Windows GUI environment and almost never use the MS-DOS shell interface. The various changes undergone by the Macintosh operating systems provides a nice study in contrast. Historically, Mac OS has not provided a command line interface, always requiring its users to interact with the operating system using its GUI. However, with the release of Mac OS X (which is in part implemented using a UNIX kernel), the operating system now provides both a new Aqua interface and command-line interface as well.

The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly 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 user programs. From the point of view of an 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 routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly), may need to be written using assembly-language instructions.

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 for the program to ask the user for the names of the two files. 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 are obtained, the program must open the input file and create the output file. Each of these operations requires another system call. There are also possible error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name or that
EXAMPLE OF STANDARD API

As an example of a standard API, consider the ReadFile() function in the Win32 API—a function for reading from a file. The API for this function appears in Figure 2.2.

A description of the parameters passed to ReadFile() is as follows:

• HANDLE file—the file to be read.
• LPVOID buffer—a buffer where the data will be read into and written from.
• DWORD bytesToRead—the number of bytes to be read into the buffer.
• LPDWORD bytesRead—the number of bytes read during the last read.
• LPOVERLAPPED ov—I indicates if overlapped I/O is being used.

function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Win32 API for Windows systems, the POSIX API for POSIX-based systems (which includes virtually all versions of UNIX, Linux, and Mac OS X), and the Java API for designing programs that run on the Java virtual machine.

Note that 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 Win32 function CreateProcess() (which unsurprisingly is used to create a new process) actually calls 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 of programming according to an API 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
• Process control
  o end, abort
  o load, execute
  o create process, terminate process
  o get process attributes, set process attributes
  o wait for time
  o wait event, signal event
  o allocate and free memory
• File management
  o create file, delete file
  o open, close
  o read, write, reposition
  o get file attributes, set file attributes
• Device management
  o request device, release device
  o read, write, reposition
  o get device attributes, set device attributes
  o logically attach or detach devices
• Information maintenance
  o get time or date, set time or date
  o get system data, set system data
  o get process, file, or device attributes
  o set process, file, or device attributes
• Communications
  o create, delete communication connection
  o send, receive messages
  o transfer status information
  o attach or detach remote devices

Figure 2.5 Types of system calls.

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 any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. In a batch system, the command interpreter usually terminates the entire job and continues with the next job.
For example, the timer construct (see Section 1.5.2) 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 insensitive to changes in policy would be more desirable. 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 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 (Section 2.7.3) 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 via user programs themselves. As an example, consider the history of UNIX. At first, it had a time-sharing scheduler. In the latest version of Solaris, the scheduling is controlled by loadable tables. Depending on the table currently loaded, the system can be time shared, batch processed, real time, fair share, or any combination. Making the scheduling mechanism general purpose allows vast policy changes to be made with a single load-new-table command. At the other extreme is a system such as Windows, in which both mechanism and policy are encoded in the system to enforce a global look and feel. All applications have similar interfaces because the interface itself is built into the kernel and system libraries. The Mac OS X operating system has similar functionality.

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.

### 2.6.3 Implementation

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++.

The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in PL/1. The Linux and Windows XP operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those accrued 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 move to some other hardware—
2.8 Virtual Machines

For the IBM VM system, a user normally runs CMS—a single-user interactive operating system. The virtual machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but it does not need to consider any user-support software. This arrangement may provide a useful way to divide the problem of designing a multiuser interactive system into two smaller pieces.

2.8.1 Implementation

Although the virtual-machine concept is useful, it is difficult to implement. Much work is required to provide an exact duplicate of the underlying machine. Remember that the underlying machine has two modes: user mode and kernel mode. The virtual-machine software can run in kernel mode, since it is the operating system. The virtual machine itself can execute in only user mode. Just as the physical machine has two modes, however, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in a physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.

Such a transfer can be accomplished as follows. When a system call, for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine. When the virtual-machine monitor gains control, it can change the register contents and program counter for the virtual machine to simulate the effect of the system call. It can then restart the virtual machine, noting that it is now in virtual kernel mode.

The major difference, of course, is time. Whereas the real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is...
Chapter 2 Operating-System Structures

spooled) or more time (because it is interpreted). In addition, the CPU is being multiprogrammed among many virtual machines, further slowing down the virtual machines in unpredictable ways. In the extreme case, it may be necessary to simulate all instructions to provide a true virtual machine. VM works for IBM machines because normal instructions for the virtual machines can execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be simulated and hence execute more slowly.

2.8.2 Benefits

The virtual-machine concept has several advantages. Notice that, in this environment, there is complete protection of the various system resources. Each virtual machine is completely isolated from all other virtual machines, so there are no protection problems. At the same time, however, there is no direct sharing of resources. Two approaches to provide sharing have been implemented. First, it is possible to share a minidisk and thus to share files. This scheme is modeled after a physical shared disk but is implemented by software. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. Again, the network is modeled after physical communication networks but is implemented in software.

Such a virtual-machine system is a perfect vehicle for operating-systems research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and it is difficult to be sure that a change in one part will not cause obscure bugs in some other part. The power of the operating system makes changing it particularly troublesome. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

The operating system, however, runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called system-development time. Since it makes the system unavailable to users, system-development time is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation seldom needs to be disrupted for system development.

2.8.3 Examples

Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming back into fashion as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: VMware and the Java virtual machine. As we will see, these virtual machines typically run on top of an operating system of any of the design types discussed earlier. Thus, operating system design methods—
remainder of the bootstrap program. All of the disk-bound bootstrap, and the
operating system itself, can be easily changed by writing new versions to disk.
A disk that has a boot partition (more on that in section 12.5.1) is called a **boot
disk** or system disk.

Now that the full bootstrap program has been loaded, it can traverse the
file system to find the operating system kernel, load it into memory, and start
its execution. It is only at this point that the system is said to be **running**.

### 2.11 Summary

Operating systems provide a number of services. At the lowest level, system
calls allow a running program to make requests from the operating system
directly. At a higher level, the command interpreter or shell provides a
mechanism for a user to issue a request without writing a program. Commands
may come from files during batch-mode execution or directly from a terminal
when in an interactive or time-shared mode. System programs are provided to
satisfy many common user requests.

The types of requests vary according to level. The system-call level must
provide the basic functions, such as process control and file and device
manipulation. Higher-level requests, satisfied by the command interpreter or
system programs, are translated into a sequence of system calls. System services
can be classified into several categories: program control, status requests, and
I/O requests. Program errors can be considered implicit requests for service.

Once the system services are provided, the structure of the operating system
will be developed. Various tables are needed to record the information that
defines the state of the computer system and the status of the system's jobs.

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. The type of
system desired is the foundation for choices among various algorithms and
strategies that will be needed.

Since an operating system is large, modularity is important. Designing a
system as a sequence of layers or using a microkernel is considered a good
technique. The virtual-machine concept takes the layered approach and treats
both the kernel of the operating system and the hardware as though they were
hardware. Even other operating systems may be loaded on top of this virtual
machine.

Throughout the entire operating-system design cycle, we must be careful
to separate policy decisions from implementation details (mechanisms). This
separation allows maximum flexibility if policy decisions are to be changed
later.

Operating systems are now almost always written in a systems-
implementation language or in a higher-level language. This feature improves
their implementation, maintenance, and portability. To create an operating
system for a particular machine configuration, we must perform system
generation.

For a computer system to begin running, the CPU must initialize and start
executing the bootstrap program in firmware. The bootstrap can execute the
operating system directly if the operating system is also in the firmware, or
it can complete a sequence in which it loads progressively smarter programs
Chapter 3 Processes

I/O request

time slice

expired

fork a
child

wait for an
interrupt

Figure 3.7 Queueing-diagram representation of process scheduling.

from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then $10/(100 + 10) = 9$ percent of the CPU is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler make a careful selection. 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. It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound
Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. 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 users may be interested in the same piece of information (for instance, a shared file), 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 elements (such as CPUs or I/O channels).

- **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.

- **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: (1) shared memory and (2) message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.13.

Both of the models just discussed 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 than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In contrast, 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 the remainder of this section, we explore each of these IPC models in more detail.
3.5 Examples of IPC Systems

3.5.1 An Example: 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.

A process must first create a shared memory segment using the `shmget()` system call (`shmget()` is derived from `SHared Memory GET`). The following example illustrates the use of `shmget()`:

```c
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

This first parameter specifies the key (or identifier) of the shared-memory segment. If this is set to `IPC_PRIVATE`, a new shared-memory segment is created. The second parameter specifies the size (in bytes) of the shared memory segment. Finally, the third parameter identifies the mode, which indicates how the shared-memory segment is to be used—that is, for reading, writing, or both. By setting the mode to `S_IRUSR | S_IWUSR`, we are indicating that the owner may read or write to the shared memory segment. A successful call to `shmget()` returns an integer identifier for the shared-memory segment. Other processes that want to use this region of shared memory must specify this identifier.

Processes that wish to access a shared-memory segment must attach it to their address space using the `shmat()` (SHared Memory ATtach) system call. The call to `shmat()` expects three parameters as well. The first is the integer identifier of the shared-memory segment being attached, and the second is a pointer location in memory indicating where the shared memory will be attached. If we pass a value of `NULL`, the operating system selects the location on the user's behalf. The third parameter identifies a flag that allows the shared-memory region to be attached in read-only or read-write mode; by passing a parameter of 0, we allow both reads and writes to the shared region.

The third parameter identifies a mode flag. If set, the mode flag allows the shared-memory region to be attached in read-only mode; if set to 0, the flag allows both reads and writes to the shared region. We attach a region of shared memory using `shmat()` as follows:

```c
shared_memory = (char *) shmat(id, NULL, 0);
```

If successful, `shmat()` returns a pointer to the beginning location in memory where the shared-memory region has been attached.

Once the region of shared memory is attached to a process's address space, the process can access the shared memory as a routine memory access using the pointer returned from `shmat()`. In this example, `shmat()` returns a pointer to a character string. Thus, we could write to the shared-memory region as follows:

```c
sprintf(shared_memory, "Writing to shared memory");
```

Other processes sharing this segment would see the updates to the shared-memory segment.

Typically, a process using an existing shared-memory segment first attaches the shared-memory region to its address space and then accesses (and possibly updates) the region of shared memory. When a process no longer requires access to the shared-memory segment, it detaches the segment from its address space using the `shmdet()` system call.
3. Do not wait at all but rather return immediately.

4. Temporarily cache a message. One message can be given to the operating system to keep, even though the mailbox to which it is being sent is full. When the message can be put in the mailbox, a message is sent back to the sender; only one such message to a full mailbox can be pending at any time for a given sending thread.

The final option is meant for server tasks, such as a line-printer driver. After finishing a request, such tasks may need to send a one-time reply to the task that had requested service; but they must also continue with other service requests, even if the reply mailbox for a client is full.

The receive operation must specify the mailbox or mailbox set from which a message is to be received. A mailbox set is a collection of mailboxes, as declared by the task, which can be grouped together and treated as one mailbox for the purposes of the task. Threads in a task can receive only from a mailbox or mailbox set for which the task has receive access. A port_status() system call returns the number of messages in a given mailbox. The receive operation attempts to receive from (1) any mailbox in a mailbox set or (2) a specific (named) mailbox. If no message is waiting to be received, the receiving thread can either wait at most $n$ milliseconds or not wait at all.

The Mach system was especially designed for distributed systems, which we discuss in Chapters 16 through 18, but Mach is also suitable for single-processor systems, as evidenced by its inclusion in the Mac OS X system. The major problem with message systems has generally been poor performance caused by double copying of messages; the message is copied first from the sender to a mailbox and then from the mailbox to the receiver. The Mach message system attempts to avoid double-copy operations by using virtual-memory-management techniques (Chapter 9). Essentially, Mach maps the address space containing the sender's message into the receiver's address space. The message itself is never actually copied. This message-management technique provides a large performance boost but works for only intrasystem messages. The Mach operating system is discussed in an extra chapter posted on our website.

3.5.3 An Example: Windows XP

The Windows XP operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows XP provides support for multiple operating environments, or subsystems, with which application programs communicate via a message-passing mechanism. The application programs can be considered clients of the Windows XP subsystem server.

The message-passing facility in Windows XP is called the local procedure-call (LPC) facility. The LPC in Windows XP communicates between two processes on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows XP. Like Mach, Windows XP uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows XP uses two types of ports: connection ports and communication ports. They
are really the same but are given different names according to how they are used. Connection ports are named objects and are visible to all processes; they give applications a way to set up communication channels (Chapter 22). The communication works as follows:

- The client opens a handle to the subsystem's connection port object.
- The client sends a connection request.
- The server creates two private communication ports and returns the handle to one of them to the client.
- The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows XP uses two types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent.

If a client needs to send a larger message, it passes the message through a section object, which sets up a region of shared memory. The client has to decide when it sets up the channel whether or not 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, but it avoids data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling.

The structure of local procedure calls in Windows XP is shown in Figure 3.17.

It is important to note that the IPC facility in Windows XP is not part of the Win32 API and hence is not visible to the application programmer. Rather,
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 subclass 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

```java
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 3.19 Date server.
3.6 Communication in Client-Server Systems

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 data 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 daemon. In addition to an IP address, an actual host name, such as www.westminstercollege.edu, can be used as well.

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 two subsections, we look at two higher-level methods of communication: remote procedure calls (RPCs) and remote method invocation (RMI).

3.6.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, which we discussed briefly in Section 3.5.2. The RPC 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 a server environment in which the processes are executed on separate systems, we must use a message-based communication scheme to provide remote service. In contrast to the IPC facility, 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 of 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 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 allow 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. Parameter marshalling involves packaging the parameters into a form that can be transmitted over
and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB, and the PCBs can be linked together to form a ready queue. Long-term (job) scheduling is the selection of processes that will be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource-allocation considerations, especially memory management. Short-term (CPU) scheduling is the selection of one process from the ready queue.

Operating systems must provide a mechanism for parent processes to create new child processes. The parent may wait for its children to terminate before proceeding, or the parent and children may execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience.

The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes require an interprocess communication mechanism to communicate with each other. Principally, communication is achieved through two schemes: shared memory and message passing. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application programmers; the operating system needs to provide only the shared memory.

The message-passing method requires the processes to exchange messages. The responsibility for providing communication rests with the operating system itself. These two schemes are not mutually exclusive and can be used simultaneously within a single operating system.

Communication in client-server systems may use (1) sockets, (2) remote procedure calls (RPCs), or (3) Java's remote method invocation (RMI). A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote application. RMI is the Java version of RPCs. RMI allows a thread to invoke a method on a remote object just as it would invoke a method on a local object. The primary distinction between RPCs and RMI is that in RPCs data are passed to a remote procedure in the form of an ordinary data structure, whereas RMI allows objects to be passed in remote method calls.

**Exercises**

1. Describe the differences among short-term, medium-term, and long-term scheduling.
2. Describe the actions taken by a kernel to context-switch between processes.
3. 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.
Project—UNIX Shell and History Feature

This project consists of modifying a C program which serves as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface provides the user a prompt after which the next command is entered. The example below illustrates the prompt `sh>` and the user's next command: `cat prog.c`. This command displays the file `prog.c` on the terminal using the UNIX `cat` command.

```
sh> 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 (i.e. `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 what is illustrated in Figure 3.11. However, UNIX shells typically also allow the child process to run in the background—concurrently—as well by specifying the ampersand (`&`) at the end of the command. By rewriting the above command as

```
sh> cat prog.c &
```

the parent and child processes now run concurrently.

The separate child process is created using the `fork()` system call and the user's command is executed by using one of the system calls in the `exec()` family (as described in Section 3.3.1).

Simple Shell

A C program that provides the basic operations of a command line shell is supplied in Figure 3.25. This program is composed of two functions: `main()` and `setup()`. The `setup()` function reads in the user's next command (which can be up to 80 characters), and then parses it into separate tokens that are used to fill the argument vector for the command to be executed. (If the command is to be run in the background, it will end with `&`, and `setup()` will update the parameter `background` so the `main()` function can act accordingly. This program is terminated when the user enters `<ControlxD>` and `setup()` then invokes `exit()`.

The `main()` function presents the prompt `COMMAND>` and then invokes `setup()`, which waits for the user to enter a command. The contents of the command entered by the user is loaded into the `args` array. For example, if the user enters `ls -1` at the `COMMAND>` prompt, `args[0]` becomes equal to the string `ls` and `args[1]` is set to the string `to -1`. (By "string", we mean a null-terminated, C-style string variable.)
The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Most modern operating systems now provide features enabling a process to contain multiple threads of control. This chapter introduces many concepts associated with multithreaded computer systems, including a discussion on APIs for the Pthreads, Win32, and Java thread libraries. We also consider many issues related to multithreaded programming and how they affect the design of modern operating systems. Finally, we explore how the Windows XP and Linux operating systems support threads at the kernel level.

**CHAPTER OBJECTIVES**

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- To discuss the APIs for Pthreads, Win32, and Java thread libraries.

### 4.1 Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, 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 (or heavyweight) 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

Many software packages that run on modern desktop PCs are multithreaded. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread...
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. The turnaround time is generally limited by the speed of the output device.

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, it is desirable 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 time-sharing systems), 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 will illustrate their operation. An accurate illustration should involve many processes, each being 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 5.7.

### 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. There are many different CPU scheduling algorithms. In this section, we describe several of them.

#### 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.

The average waiting time under the FCFS policy, however, 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:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>
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:

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 process's 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.

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 time-sharing systems, where it is important that each user 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:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>6</td>
</tr>
<tr>
<td>P_2</td>
<td>8</td>
</tr>
<tr>
<td>P_3</td>
<td>7</td>
</tr>
<tr>
<td>P_4</td>
<td>3</td>
</tr>
</tbody>
</table>

Using SJF scheduling, we would schedule these processes according to the
following Gantt chart:

```
   P_4  |  P_1  |  P_3  |  P_2

  0    3    9    16   24
```

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.

The real difficulty with the SJF algorithm is knowing the length of the next
CPU request. For long-term (job) scheduling in a batch system, we can use as
the length the process time limit that a user specifies when he submits the
job. Thus, users are motivated to estimate the process time limit accurately,
since a lower value may mean faster response. (Too low a value will cause
a time-limit-exceeded error and require resubmission.) SJF scheduling is used
frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level
of short-term CPU scheduling. There is no way to know the length of the next
CPU burst. One approach 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.
Thus, 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. Let t_n be the length of the n-th CPU
will preempt the currently executing process, whereas a nonpreemptive 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:

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

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:

![Gantt chart](image)

Process P₁ is started at time 0, since it is the only process in the queue. Process P₂ arrives at time 1. The remaining time for process P₁ (7 milliseconds) is larger than the time required by process P₂ (4 milliseconds), so process P₁ is preempted, and process P₂ is scheduled. The average waiting time for this example is: 

\[ \frac{(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))}{4} = \frac{26}{4} = 6.5 \text{ milliseconds}. \]

Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

### 5.3.3 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 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 P₁, P₂, • • •, P₅, with the length of the CPU burst given in milliseconds:

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>P₅</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

...
Threads in the real-time class are given the highest priority. This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time. A real-time process will run before a process in any other class. In general, however, few processes belong to the real-time class.

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. As mentioned, Solaris has traditionally used the many-to-many model (4.2.3) but with Solaris 9 switched to the one-to-one model (4.2.2).

5.6.2 Example: Windows XP Scheduling

Windows XP schedules threads using a priority-based, preemptive scheduling algorithm. The Windows XP scheduler ensures that the highest-priority thread will always run. The portion of the Windows XP 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, it terminates, until its time quantum ends, or until it can execute a 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 XP kernel and the Win32 API. The Win32 API identifies several priority classes to which a process can belong. These include:

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- IDLE_PRIORITY_CLASS

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 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 files or messages. The former case is achieved through the use of lightweight processes or threads, which we discussed in Chapter 4. Concurrent access to shared data may result in data inconsistency. 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

• To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
• To present both software and hardware solutions of the critical-section problem.
• To introduce the concept of atomic transaction and describe mechanisms to ensure atomicity.

6.1 Background

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 representative of operating systems. Specifically, in Section 3.4.1, we described how a bounded buffer could be used to enable processes to share memory.

Let us return to our consideration of the bounded buffer. As we pointed out, our solution allows at most \( \text{BUFFER\_SIZE} - 1 \) 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 counter, initialized to 0. Counter is incremented every time we add a new item to the buffer and is decremented
do{
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);

Figure 6.3 Solution to the critical-section problem using locks.

6.4 Synchronization Hardware

We have just described one software-based solution to the critical-section problem. In general, we can state that any solution to the critical-section problem requires a simple tool—a lock. Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated in Figure 6.3.

In the following discussions, we explore several more solutions to the critical-section problem, using techniques ranging from hardware to software-based APIs available to application programmers. All these solutions are based on the premise of locking; however, as we shall see, the design of such locks can be quite sophisticated.

Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence 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. This is the approach taken by nonpreemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the

```c
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Figure 6.4 The definition of the TestAndSet() instruction.
wait. The variable \( \text{waiting}[i] \) can become false only if another process leaves its critical section; only one \( \text{waiting}[i] \) is set to \text{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 \text{false} or sets \( \text{waiting}[j] \) to \text{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 \( \text{waiting} \) in the cyclic ordering \( \{z' + 1, i + 2, ..., n - 1, 0, ..., i - 1\} \). It designates the first process in this ordering that is in the entry section (\( \text{waiting}[j] \) = \text{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.

Unfortunately for hardware designers, implementing atomic \text{TestAndSet}() instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

### 6.5 Semaphores

The various hardware-based solutions to the critical-section problem (using the \text{TestAndSet}() and \text{Swap}() instructions) presented in Section 6.4 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.

A semaphore \( S \) is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: \text{wait()} and \text{signal()}. The \text{wait()} operation was originally termed \text{P} (from the Dutch \text{proberen}, "to test"); \text{signal()} was originally called \text{V} (from \text{verhogen}, "to increment"). The definition of \text{wait()} is as follows:

```c
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

The definition of \text{signal()} is as follows:

```c
signal(S) {
    S++;
}
```

All the modifications to the integer value of the semaphore in the \text{wait()} and \text{signal()} operations must be executed indivisibly. 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 \text{wait}(S), the testing of the integer value of \( S \) (\( S \leq 0 \)), and its possible modification (\( S-- \)), must also be executed without interruption. We shall see how these operations can be implemented in Section 6.5.2; first, let us see how semaphores can be used.
6.5.2 Implementation

The main disadvantage of the semaphore definition 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 entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)

To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations. 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 **block** itself. The block 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 blocked, waiting on a semaphore `S`, should be restarted when some other process executes a `signal()` operation. The process is awakened 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 a "C" struct:

```c
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.

The `wait()` semaphore operation can now be defined as

```c
wait(semaphore *S) {
    S->value--;  
    if (S->value < 0) {
        add this process to S->list;  
        block();  
    }
}
```
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.

6.5.3 Deadlocks and Starvation

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, we consider a system consisting of two processes, \( P_0 \) and \( P_1 \), each accessing two semaphores, \( S \) and \( Q \), set to the value 1:

\[
\begin{align*}
P_0 & \quad P_1 \\
& \text{wait}(S); \quad \text{wait}(Q); \\
& \text{wait}(Q); \quad \text{wait}(S); \\
& \text{signal}(S); \quad \text{signal}(Q); \\
& \text{signal}(Q); \quad \text{signal}(S);
\end{align*}
\]

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 deadlock 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 resource acquisition and release. However, other types of events may result in deadlocks, as we shall show in Chapter 7. In that chapter, we shall describe various mechanisms for dealing with the deadlock problem.

Another problem related to deadlocks is indefinite blocking, or starvation, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

6.6 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.
• **Shared.** If a transaction $T_i$ has obtained a shared-mode lock (denoted by $S$) on data item $Q$, then $T_i$ can read this item but cannot write $Q$.

• **Exclusive.** If a transaction $T_i$ has obtained an exclusive-mode lock (denoted by $X$) on data item $Q$, then $T_i$ can both read and write $Q$.

We require that every transaction request a lock in an appropriate mode on data item $Q$, depending on the type of operations it will perform on $Q$.

To access data item $Q$, transaction $T_j$ must first lock $Q$ in the appropriate mode. If $Q$ is not currently locked, then the lock is granted, and $T_i$ can now access it. However, if the data item $Q$ is currently locked by some other transaction, then $T_j$ may have to wait. More specifically, suppose that $T_j$ requests an exclusive lock on $Q$. In this case, $T_j$ must wait until the lock on $Q$ is released. If $T_i$ requests a shared lock on $Q$, then $T_j$ must wait if $Q$ is locked in exclusive mode. Otherwise, it can obtain the lock and access $Q$. Notice that this scheme is quite similar to the readers-writers algorithm discussed in Section 6.6.2.

A transaction may unlock a data item that it locked at an earlier point. It must, however, hold a lock on a data item as long as it accesses that item. Moreover, it is not always desirable for a transaction to unlock a data item immediately after its last access of that data item, because serializability may not be ensured.

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

- **Growing phase.** A transaction may obtain locks but may not release any lock.
- **Shrinking phase.** A transaction may release locks but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued.

The two-phase locking protocol ensures conflict serializability (Exercise 6.25). It does not, however, ensure freedom from deadlock. In addition, it is possible that, for a given set of transactions, there are conflict-serializable schedules that cannot be obtained by use of the two-phase locking protocol. However, to improve performance over two-phase locking, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data.

### 6.9.4.3 Timestamp-Based Protocols

In the locking protocols described above, the order followed by pairs of conflicting transactions is determined at execution time by the first lock that both request and that involves incompatible modes. Another method for determining the serializability order is to select an order in advance. The most common method for doing so is to use a **timestamp** ordering scheme.

With each transaction $T_i$ in the system, we associate a unique fixed timestamp, denoted by $TS(T_i)$. This timestamp is assigned by the system...
Exercises 235

6.27 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 —once finished—will return them. 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 requests will only be granted 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:

```c
#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:

```c
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return */
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:

```c
/* 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, fix the race condition.
6.28 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 obtain a number of resources:

```c
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

```c
decrease_count(count);
```

The process will only return from this function call when sufficient resources are available.

Project: Producer-Consumer Problem

In Section 6.6.1, we present a semaphore-based solution to the producer-consumer problem using a bounded buffer. In this project, we will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 6.10 and 6.11. The solution presented in Section 6.6.1 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, standard counting semaphores will be used for empty and full, and, rather than a binary semaphore, a mutex lock will be used to represent mutex. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with these empty, full, and mutex structures. You can solve this problem using either Pthreads or the Win32 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:

```c
/* 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 as:
In Section 6.5, we described the classical wait() and signal() semaphore operations. Pthreads names the wait() and signal() operations sem_wait() and sem_post(), respectively. The code example below creates a binary semaphore mutex with an initial value of 1 and illustrates its use in protecting a critical section:

```c
#include <semaphore.h>
sem_t sem_mutex;

/* create the semaphore */
sem_init(&mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&mutex);

/*** critical section ***/

/* release the semaphore */
sem_post(&mutex);
```

Win32

Details concerning thread creation using the Win32 API are available in Chapter 4. Please refer to that chapter for specific instructions.

Win32 Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 6.8.2. The following illustrates how to create a mutex lock using the CreateMutex() function:

```c
#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 are disallowing any children of the process creating this mutex lock to inherit the handle of the mutex. The second parameter indicates whether the creator of the mutex is the initial owner of the mutex lock. 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 naming of 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 6.8.2, we identified dispatcher objects as being either signaled or nonsignaled. A signaled object is available for ownership; once a dispatcher object (such as a mutex lock) is acquired, it moves to the nonsignaled state. When the object is released, it returns to signaled.
4. **Circular wait.** A set \( \{ P_0, P_1, ..., P_n \} \) of waiting processes must exist such that:

- \( P_0 \) is waiting for a resource held by \( P_1 \),
- \( P_1 \) is waiting for a resource held by \( P_2 \),
- \( ..., \)
- \( P_n \) is waiting for a resource held by \( P_0 \).

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 conditions are not completely independent. We shall see in Section 7.4, however, that it is useful to consider each condition separately.

### 7.2.2 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: \( P = \{ P_1, P_2, ..., P_n \} \), the set consisting of all the active processes in the system, and \( R = \{ R_1, R_2, ..., R_m \} \), the set consisting of all resource types in the system.

A directed edge from process \( P_i \) to resource type \( R_j \) is denoted by \( P_i \rightarrow R_j \); it signifies that process \( P_i \) 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 process \( P_i \) is denoted by \( R_j \rightarrow P_i \); it signifies that an instance of resource type \( R_j \) has been allocated to process \( P_i \). A directed edge \( P_i \rightarrow R_j \) is called a **request edge**; a directed edge \( R_j \rightarrow P_i \) is called an **assignment edge**.

Pictorially, we represent each process \( P_i \) as a circle and each resource type \( R_j \) as a rectangle. 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 to only the rectangle \( R_j \), whereas an assignment edge must also designate one of the dots in the rectangle.

When process \( P_i \) 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 process 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 7.2 depicts the following situation.

- **The sets \( P, R, \) and \( E \):**
  - \( P = \{ P_1, P_2, P_3 \} \)
  - \( R = \{ R_1, R_2, R_3, R_4 \} \)
  - \( E = \{ P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_3 \} \)

- **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 \( R_4 \).
process $P_2$ to release resource $R_2$. In addition, process $P_2$ is waiting for process $P_2$ to release resource $R_1$.

Now consider the resource-allocation graph in Figure 7.4. In this example, we also have a cycle

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$$

However, there is no deadlock. Observe that process $P_4$ may release its instance of resource type $R_2$. That resource can then be allocated to $P_3$, 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.
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. In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods, which must be used constantly. Also, in some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority (or any process 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.

7.4 Deadlock Prevention

As we noted in Section 7.2.1, 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.

7.4.1 Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously held by several processes.Sharable resources, in contrast, 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 processes attempt to open a read-only file at the same time, then they are granted simultaneous access to the file. A process 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.

7.4.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases
both the DVD drive and the disk file. The process must then again request the
disk file and the printer. After copying the disk file to the printer, it releases
these two resources and terminates.

Both these protocols have two main disadvantages. First, resource utiliza-
tion may be low, since resources may be allocated but unused for a long period.
In the example given, for instance, we can release the DVD drive and disk file,
and then again request the disk file and printer, only if we can be sure that our
data will remain on the disk file. If we cannot be assured that they will, then
we must request all resources at the beginning for both protocols.

Second, starvation is possible. A process 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 process.

7.4.3 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 process is holding some
resources and requests another resource that cannot be immediately allocated
to it (that is, the process must wait), then all resources currently being held
are preempted. In other words, the resources are implicitly released. The
preempted resources are added to the list of resources for which the process is
waiting. The process will be restarted only when it can regain its old resources,
as well as the new ones that it is requesting.

Alternatively, if a process 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 needed by a waiting process that is waiting for additional
resources. If so, we preempt the desired resources from the waiting process and
allocate them to the requesting process. If the resources are neither available
nor held by a waiting process, the requesting process must wait. While it is
waiting, some of its resources may be preempted, but only if another process
requests them. A process 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 memory space. It cannot generally
be applied to such resources as printers and tape drives.

7.4.4 Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One
way to ensure that this condition never holds is to impose a total ordering of
all resource types and to require that each process requests resources in an
increasing order of enumeration.

To illustrate, we let $R = \{R_1, R_2, ..., R_n\}$ be the set of resource types. We
assign to each resource type a unique integer number, which allows us to
cmpare 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. For example, if the set of resource types $R$ includes
an unsafe state. Therefore, process $P_i$ will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 7.6. Suppose that $P_2$ requests $R_2$. Although $R_2$ is currently free, we cannot allocate it to $P_2$, since this action will create a cycle in the graph (Figure 7.7). A cycle indicates that the system is in an unsafe state. If $P_1$ requests $R_2$, and $P_2$ requests $R_1$, then a deadlock will occur.

7.5.3 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The 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 process 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 process must wait until some other process 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. Let $n$ be the number of processes in the system and $m$ be the number of resource types. We need the following data structures:

- **Available.** A vector of length $m$ indicates the number of available resources of each type. If $Available[j]$ equals $k$, there are $k$ instances of resource type $R_j$ available.
- **Max.** An $n \times m$ matrix defines the maximum demand of each process. If $Max[i][j]$ equals $k$, then process $P_i$ 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 process. If $\text{Allocation}[i][j]$ equals $k$, then process $P_i$ is currently allocated $k$ instances of resource type $R_j$.

• Need. An $n \times m$ matrix indicates the remaining resource need of each process. If $\text{Need}[i][j]$ equals $k$, then process $P_i$ 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 $i = 1, 2, ..., n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices $\text{Allocation}$ and $\text{Need}$ as vectors and refer to them as $\text{Allocation}_i$ and $\text{Need}_i$. The vector $\text{Allocation}_i$ specifies the resources currently allocated to process $P_i$; the vector $\text{Need}_i$ specifies the additional resources that process $P_i$ may still request to complete its task.

7.5.3.1 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 $\text{Work}$ and $\text{Finish}$ be vectors of length $m$ and $n$, respectively. Initialize $\text{Work} = \text{Available}$ and $\text{Finish}[i] = \text{false}$ for $i = 0, 1, ..., n-1$.
2. Find an $i$ such that both
   a. $\text{Finish}[i] = \text{false}$
   b. $\text{Need}_i \leq \text{Work}$

   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 $m \times n$ operations to determine whether a state is safe.

7.5.3.2 Resource-Request Algorithm

We now describe the algorithm which determines if requests can be safely granted.

Let $\text{Request}_i$ be the request vector for process $P_i$. If $\text{Request}_i[j] = k$, then process $P_i$ wants $k$ instances of resource type $R_j$. When a request for resources is made by process $P_i$, 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 process has exceeded its maximum claim.
2. If \( \text{Request}_i \leq \text{Available} \), go to step 3. Otherwise, \( P_i \) must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process \( P_i \) by modifying the state as follows:

\[
\begin{align*}
\text{Available} &= \text{Available} - \text{Request}_i; \\
\text{Allocation}_i &= \text{Allocation}_i + \text{Request}_i; \\
\text{Need}_i &= \text{Need}_i - \text{Request}_i;
\end{align*}
\]

If the resulting resource-allocation state is safe, the transaction is completed, and process \( P_i \) is allocated its resources. However, if the new state is unsafe, then \( P_i \) must wait for \( \text{Request}_i \), and the old resource-allocation state is restored.

### 7.5.3.3 An Illustrative Example

Finally, to illustrate the use of the banker’s algorithm, consider a system with five processes \( P_0 \) through \( P_4 \) and three resource types \( A, B, \) and \( C \). Resource type \( A \) has 10 instances, resource type \( B \) has 5 instances, and resource type \( C \) has 7 instances. Suppose that, at time \( T_o \), the following snapshot of the system has been taken:

<table>
<thead>
<tr>
<th></th>
<th>( P_0 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
<th>( \text{Max} )</th>
<th>( \text{Available} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>( B )</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>( C )</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>33</td>
</tr>
</tbody>
</table>

The content of the matrix \( \text{Need} \) is defined to be \( \text{Max} - \text{Allocation} \) and is as follows:

\[
\begin{align*}
\text{Need} \\
A & | B & C \\
\hline
P_0 & 7 & 4 & 3 \\
P_1 & 1 & 2 & 2 \\
P_2 & 6 & 0 & 0 \\
P_3 & 0 & 1 & 1 \\
P_4 & 4 & 3 & 1
\end{align*}
\]

We claim that the system is currently in a safe state. Indeed, the sequence \( <P_0, P_3, P_4, P_2, P_0> \) satisfies the safety criteria. Suppose now that process \( P_1 \) 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:
Suppose now that process \( P_2 \) makes one additional request for an instance of type \( C \). The Request matrix is modified as follows:

\[
\begin{array}{ccc}
& A & BC \\
\hline
P_0 & 0 & 00 \\
P_1 & 2 & 02 \\
P_2 & 0 & 01 \\
P_3 & 1 & 00 \\
P_4 & 0 & 02 \\
\end{array}
\]

We claim that the system is now deadlocked. Although we can reclaim the resources held by process \( P_0 \), the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes \( P_1, P_2, P_3, \) and \( P_4 \).

### 7.6.3 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 processes will be affected by deadlock when it happens?**

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, 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 processes but also the specific process that "caused" the deadlock. (In reality, each of the deadlocked processes 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 process.

Of course, if the deadlock-detection algorithm is invoked for every resource request, this will incur a considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at less frequent 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, there may be many cycles in the resource graph. In this case, we would generally not be able to tell which of the many deadlocked processes "caused" the deadlock.
7.7.2 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 preemption 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 during its execution.

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 a 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 that must be dealt with in any practical system. 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.

7.8 Summary

A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. There are three principal methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, ensuring that the system, will never enter a deadlock state.

- Allow the system to enter a deadlock state, detect it, and then recover.

- Ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows.
7.4 Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:

a. Runtime overheads
b. System throughput

7.5 In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase Available (new resources added).
b. Decrease Available (resource permanently removed from system).
c. Increase Max for one process (the process needs more resources than allowed; it may want more).
d. Decrease Max for one process (the process decides it does not need that many resources).
e. Increase the number of processes.
f. Decrease the number of processes.

7.6 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock free.
8.1.3 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 the **memory-address register** of the memory—is commonly referred to as a **physical address**.

The compile-time and load-time address-binding methods generate 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). We can choose from many different methods to accomplish such mapping, as we discuss in
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; compaction 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 the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging (Section 8.4) and segmentation (Section 8.6). These techniques can also be combined (Section 8.7).

### 8.4 Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found

![Figure 8.7 Paging hardware.](image-url)
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 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 to only 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.

8.4.4 Shared Pages

An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code), however, it can be shared, as shown in Figure 8.13. Here we see a three-page editor—each page 50 KB in size (the large page size is used to simplify the figure)—being shared among three processes. Each process has its own data page.

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 editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings.

Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. 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.

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
Figure 8.15 Address translation for a two-level 32-bit paging architecture.

table in this manner, the operating system can leave partitions unused until a process needs them. An address on the VAX architecture is as follows:

\[
\begin{array}{ccc}
\text{section} & \text{page} & \text{offset} \\
\hline
s & p & d \\
\end{array}
\]

where \(s\) designates the section number, \(p\) is an index into the page table, and \(d\) is the displacement within the page. Even when this scheme is used, the size of a one-level page table for a VAX process using one section is \(2^{21}\) bits + 4 bytes per entry = 8 MB. So that main-memory usage is reduced further, the VAX pages the user-process page tables.

For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us 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:

\[
\begin{array}{ccc}
\text{outer page} & \text{inner page} & \text{offset} \\
\hline
p_1 & p_2 & d \\
\end{array}
\]

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. 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); a 64-bit address space is still daunting:

\[
\begin{array}{cccc}
\text{2nd outer page} & \text{outer page} & \text{inner page} & \text{offset} \\
\hline
p_1 & p_2 & p_3 & d \\
\end{array}
\]

The outer page table is still \(2^{34}\) bytes in size.
Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

### 8.6.2 Hardware

Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a segment table. Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 8.19. A logical address consists of two parts: a segment number, $s$, and an offset into that segment, $d$. The segment number is used as an index to the segment table. The offset $d$ of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

As an example, consider the situation shown in Figure 8.20. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of the segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped...
8.7 Example: The Intel Pentium

Both paging and segmentation have advantages and disadvantages. In fact, some architectures provide both. In this section, we discuss the Intel Pentium architecture, which supports both pure segmentation and segmentation with paging. We do not give a complete description of the memory-management structure of the Pentium in this text. Rather, we present the major ideas on which it is based. We conclude our discussion with an overview of Linux address translation on Pentium systems.

In Pentium systems, 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 8.21.

8.7.1 Pentium Segmentation

The Pentium architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 KB. The logical-address space
The segments for user code and user data are shared by all processes running in user mode. This is possible because all processes use the same logical address space and all segment descriptors are stored in the global descriptor table (GDT). Furthermore, each process has its own task-state segment (TSS), and the descriptor for this segment is stored in the GDT. The TSS is used to store the hardware context of each process during context switches. The default LDT segment is normally shared by all processes and is usually not used. However, if a process requires its own LDT, it can create one and use that instead of the default LDT.

As noted, each segment selector includes a 2-bit field for protection. Thus, the Pentium allows four levels of protection. Of these four levels, Linux only recognizes two: user mode and kernel mode.

Although the Pentium uses a two-level paging model, Linux is designed to run on a variety of hardware platforms, many of which are 64-bit platforms where two-level paging is not plausible. Therefore, Linux has adopted a three-level paging strategy that works well for both 32-bit and 64-bit architectures.

The linear address in Linux is broken into the following four parts:

```
global directory  middle directory  page table  offset
```

Figure 8.24 highlights the three-level paging model in Linux.

The number of bits in each part of the linear address varies according to architecture. However, as described earlier in this section, the Pentium architecture only uses a two-level paging model. How, then, does Linux apply its three-level model on the Pentium? In this situation, the size of the middle directory is zero bits, effectively bypassing the middle directory.
In Chapter 8, 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. The 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 user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files easily 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 discuss virtual memory in the form of demand paging and examine its complexity and cost.

CHAPTER OBJECTIVES

- To describe the benefits of a virtual memory system.
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.
- To discuss the principles of the working-set model.

9.1 Background

The memory-management algorithms outlined in Chapter 8 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 loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.
• Because each user 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 each user program into memory, so each user program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and the user.

Virtual memory involves the separation of logical memory as perceived by users 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 9.1). 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 the problem to be programmed.

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 9.2. Remember from Chapter 8, though, that in fact physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory-management unit (MMU) to map logical pages to physical page frames in memory.

Note in Figure 9.2 that we allow for 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

![Figure 9.2 Virtual address space.](image)
9.2.1 Basic Concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid-invalid bit scheme described in Section 8.5 can be used for this purpose. This time, however, when this 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 on the disk. 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 either simply marked, invalid or contains the address of the page on disk. This situation is depicted in Figure 9.5.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes, and accesses pages that are memory resident, execution proceeds normally.

![Figure 9.5 Page table when some pages are not in main memory.](image-url)
Recall that the fork() system call creates a child process as 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. Alternatively, 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 attempts to modify a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures 9.7 and Figure 9.8, 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 then create a copy 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.
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. Those 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.

9.6.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging (Section 9.9.1), 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 9.21). If the actual page-fault rate exceeds the upper limit, we allocate the process 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 suspend a process. If the page-fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page-fault rates.

Figure 9.21 Page-fault frequency.
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 (comprised 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 exist, 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 comprised 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 where 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.
of free memory at \textit{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 \textit{minfree}, the pageout process is called for every request for a new page.

Recent releases of the Solaris kernel have provided enhancements of the paging algorithm. One such enhancement involves recognizing pages from shared libraries. Pages belonging to libraries that are being shared by several processes—even if they are eligible to be claimed by the scanner—are skipped during the page-scanning process. Another enhancement concerns distinguishing pages that have been allocated to processes from pages allocated to regular files. This is known as priority paging and is covered in Section 11.6.2.

9.11 Summary

It is desirable to be able to execute a process whose logical address space is larger than the available physical address space. Virtual memory is a technique that enables us to map a large logical address space onto a smaller physical memory. Virtual memory allows us to run extremely large processes and to raise the degree of multiprogramming, increasing CPU utilization. Further, it frees application programmers from worrying about memory availability. In addition, within the memory, several processes can share system libraries and files. Virtual memory also enables us to use an efficient type of process creation known as copy-on-write, wherein parent and child processes share actual pages of memory.

Virtual memory is commonly implemented by demand paging. Pure demand paging never brings in a page until that page is referenced. The first reference causes a page fault to the operating system. The operating-system kernel consults an internal table to determine where the page is located on the backing store. Then finds a free frame and reads the page in from the backing store. The page table is updated to reflect this change, and the instruction that caused the page fault is restarted. This approach allows a process to run even though its entire memory image is not in main memory at once. As long as the page-fault rate is reasonably low, performance is acceptable.

We can use demand paging to reduce the number of frames allocated to a process. This arrangement can increase the degree of multiprogramming (allowing more processes to be available for execution at one time) and—in theory, at least—the CPU utilization of the system. It also allows processes to be run even though their memory requirements exceed the total available physical memory. Such processes run in virtual memory.

If total memory requirements exceed the physical memory, then it may be necessary to replace pages from memory to free frames for new pages. Various page-replacement algorithms are used. FIFO page replacement is easy to program but suffers from Belady's anomaly. Optimal page replacement requires future knowledge. LRU replacement is an approximation of optimal page replacement, but even it may be difficult to implement. Most page-replacement algorithms, such as the second-chance algorithm, are approximations of LRU replacement.
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, alphanumerical, 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.

The information in a file is defined by its creator. Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined structure, which depends on its type. A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

### 10.1.1 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 floppy disk, send it in an e-mail, or copy it across a network, and it could still be called example.c on the destination system.

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.
- **Identifier.** 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.
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.

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. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry. The `open count` tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.

- **Disk location of the file.** Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk 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 LOCKING IN JAVA**

In the Java API, acquiring a lock requires first obtaining the `FileChannel` for the file intended to be locked. The lock method 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 a shared lock; 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 10.1 illustrates file locking in Java. This program acquires two locks on the file `file.txt`. The first half of the file is acquired as an exclusive lock; the lock for the second half is a shared lock.
beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block 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 discussed in Chapter 11) 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 \( L \times (N) \) 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 10.4. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

### 10.2.3 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.

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 would contain pointers to secondary index files, which would 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 block points 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 10.5 shows a similar situation as implemented by VMS index and relative files.

10.3 Directory Structure

Up to this point, we have been discussing "a file system." In reality, systems may have zero or more file systems, and the file systems may be of varying types. For example, a typical Solaris system may have a few UFS file systems, a VFS file system, and some NFS file systems. The details of file system implementation are found in Chapter 11.

The file systems of computers, then, can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization involves the use of directories. In this section, we explore the topic of directory structure. First, though, we explain some basic features of storage structure.

10.3.1 Storage Structure

A disk (or any storage device that is large enough) can be used in its entirety for a file system. Sometimes, though, it is desirable to place multiple file systems
their own directories. The common subdirectory should be shared. A shared directory or file will exist 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 cycle—allows directories to share subdirectories and files (Figure 10.10). The same file or subdirectory may be in two different directories. An 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 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 UFD of each team member will 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 many of the 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 their having a special type on systems that support types) and are
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 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.

10.4 File-System Mounting

Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure can be built out of multiple 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. 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 as appropriate.

To illustrate file mounting, consider the file system depicted in Figure 10.12, where the triangles represent subtrees of directories that are of interest. Figure 10.12(a) shows an existing file system, while Figure 10.12(b) shows an unmounted volume residing on /device/dsk. At this point, only the files on the existing file system can be accessed. Figure 10.13 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 10.12.

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
11.3.1 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, 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, or with 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 disk. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted makes creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated data structure, such as a B-tree, might be used. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

11.3.2 Hash Table

Another data structure used for a file directory is a hash table. With this method, 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, a chained-overflow hash table can be used. 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.
Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk 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 11.5. 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 8.3, 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 disk space is broken into little pieces. External fragmentation exists whenever the 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 file. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

Some older PC systems used contiguous allocation on floppy disks. To prevent loss of significant amounts of disk space to external fragmentation, the user had to run a repacking routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk 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. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space 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.
addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

- **Multilevel index.** A variant of the linked representation is to use 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 4-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 the UFS, 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 indirect block contains the address of a triple indirect block. 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 implementations, including Solaris and IBM's AIX, now support up to 64-bit file pointers. Pointers of this size allow files and file systems to be terabytes in size. A UNIX *inode* is shown in Figure 11.9.

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.

### 11.4.4 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 disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk 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* disk reads. This
For instance, the version of the UNIX operating system from Sun Microsystems was changed in 1991 to improve performance in the file-system allocation algorithm. The performance measurements indicated that the maximum disk throughput on a typical workstation (a 12-MIPS SPARCstation) took 50 percent of the CPU and produced a disk bandwidth of only 1.5 MB per second. To improve performance, Sun made changes to allocate space in clusters of 56 KB whenever possible (56 KB was the maximum size of a DMA transfer on Sun systems at that time). This allocation reduced external fragmentation, and thus seek and latency times. In addition, the disk-reading routines were optimized to read in these large clusters. The inode structure was left unchanged. As a result of these changes, plus the use of read-ahead and free-behind (discussed in Section 11.6.2), 25 percent less CPU was used, and throughput substantially improved.

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 reasonably could be used to optimize head movements.

11.5 Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files if possible. (Write-once optical disks only allow one write to any given sector, and thus such 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 disk 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 disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we discuss next.

11.5.1 Bit Vector

Frequently, the free-space list is implemented as a bit map 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 bit map would be

```
00111110011111000011000001100000 ... 
```

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. For example, the Intel family starting with the 80386 and the Motorola family starting with the 68020 (processors that have powered PCs and Macintosh systems, respectively) have instructions that return the offset in a word of the first bit with the value 1. One technique
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, UNIX caches directory entries for reads; but any data 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.

11.7.2 Backup and Restore

Magnetic disks 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 disk to another storage device, such as a floppy disk, magnetic tape, optical disk, or other hard disk. Recovery from the loss of an individual file, or of an entire disk, 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 therefore be as follows:

- Day 1. Copy to a backup medium all files from the disk. 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. In this manner, we can restore an entire disk 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 tapes or disks 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 medium needed and the number of days back from which a restore can be done. 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...
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 \(M_1\)—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 \(M_1\) can access files in the remote directory in a totally transparent manner.

To illustrate file mounting, consider the file system depicted in Figure 11.13, where the triangles represent subtrees of directories that are of interest. The figure shows three independent file systems of machines named \(U\), \(S_1\), and \(S_2\). At this point, at each machine, only the local files can be accessed. In Figure 11.14(a), the effects of mounting \(S_1:/usr/shared\) over \(U:/usr/local\) are shown. This figure depicts the view users on \(U\) have of their file system. Notice that 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. 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.
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 re-validate 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. In contrast to the system used in Sprite distributed file system, delayed-write is retained even when a file is opened concurrently, in conflicting modes. Hence, UNIX semantics Section 10.5.3.1) 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, even if 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 10.5.2). In spite of these drawbacks, the utility, speed, and performance of this mechanism make it the most widely used multi-vendor-distributed file system in operation.

11.10 Example: The WAFL File System

Disk I/O has a huge impact on system performance. As a result, 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 WAFL file system from Network Appliance is an example of this sort of optimization. WAFL, the write-anywhere file layout, is a powerful, elegant file system optimized for random writes.

WAFL is used exclusively on network file servers produced by Network Appliance and so is meant for use as a distributed file system. It can provide files to clients via the NFS, CIFS, 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.
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 the damage. Operating-system backup tools allow disk data to be copied to tape, enabling the user to recover from data or even disk loss due to hardware failure, operating system fault, or user error.

Network file systems, such as NFS, use client-server methodology to allow users to access files and directories from remote machines as if they were on local file systems. System calls on the client are translated into network protocols and retranslated into file-system operations on the server. Networking and multiple-client access create challenges in the areas of data consistency and performance.

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.

Exercises

11.1 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?

a. All extents are of the same size, and the size is predetermined.

b. Extents can be of any size and are allocated dynamically.

c. Extents can be of a few fixed sizes, and these sizes are predetermined.

11.2 What are the advantages of the variant of linked allocation that uses a FAT to chain together the blocks of a file?

11.3 Consider a system where free space is kept in a free-space list.

a. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.
12.2 Disk Structure

Modern disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size, such as 1,024 bytes. This option is described in Section 12.5.1. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives.

Let's look more closely at the second reason. On media that use constant linear velocity (CLV), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as constant angular velocity (CAV).

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.
Whether the storage medium is a removable magnetic disk, a DVD, or a magnetic tape, the operating system needs to provide several capabilities to use removable media for data storage. These capabilities are discussed in Section 12.9.2.

12.9.2 Operating-System Support

Two major jobs of an operating system are to manage physical devices and to present a virtual machine abstraction to applications. In this chapter, we have seen that, for hard disks, the operating system provides two abstractions. One is the raw device, which is just an array of data blocks. The other is a file system. For a file system on a magnetic disk, the operating system queues and schedules the interleaved requests from several applications. Now, we shall see how the operating system does its job when the storage media are removable.

12.9.2.1 Application Interface

Most operating systems can handle removable disks almost exactly as they do fixed disks. When a blank cartridge is inserted into the drive (or mounted), the cartridge must be formatted, and then an empty file system is generated on the disk. This file system is used just like a file system on a hard disk.

Tapes are often handled differently. The operating system usually presents a tape as a raw storage medium. An application does not open a file on the tape; it opens the whole tape drive as a raw device. Usually, the tape drive then is reserved for the exclusive use of that application until the application exits or closes the tape device. This exclusivity makes sense, because random access on a tape can take tens of seconds, or even a few minutes, so interleaving random accesses to tapes from more than one application would be likely to cause thrashing.

When the tape drive is presented as a raw device, the operating system does not provide file-system services. The application must decide how to use the array of blocks. For instance, a program that backs up a hard, disk to tape might store a list of file names and sizes at the beginning of the tape and then copy the data of the files to the tape in that order.

It is easy to see the problems that can arise from this way of using tape. Since every application makes up its own rules for how to organize a tape, a tape full of data can generally be used by only the program that created it. For instance, even if we know that a backup tape contains a list of file names and file sizes followed by the file data in that order, we still would find it difficult to use the tape. How exactly are the file names stored? Are the file sizes in binary or in ASCII? Are the files written one per block, or are they all concatenated together in one tremendously long string of bytes? We do not even know the block size on the tape, because this variable is generally one that can be chosen separately for each block written.

For a disk drive, the basic operations are read(), write(), and seek(). Tape drives have a different set of basic operations. Instead of seek(), a tape drive uses the locate() operation. The tape locate() operation is more precise than the disk seek() operation, because it positions the tape to a specific logical block, rather than an entire track. Locating to block 0 is the same as rewinding the tape.
For most kinds of tape drives, it is possible to locate to any block that has
been written on a tape. In a partly filled tape, however, it is not possible to
locate into the empty space beyond the written area, because most tape drives
do not manage their physical space in the same way disk drives do. For a disk
drive, the sectors have a fixed size, and the formatting process must be used to
place empty sectors in their final positions before any data can be written. Most
tape drives have a variable block size, and the size of each block is determined
on the fly when that block is written. If an area of defective tape is encountered
during writing, the bad area is skipped and the block is written again. This
operation explains why it is not possible to locate into the empty space beyond
the written area—the positions and numbers of the logical blocks have not yet
been determined.

Most tape drives have a `read_position()` operation that returns the
logical block number where the tape head is. Many tape drives also support a
`space()` operation for relative motion. So, for example, the operation `space(-2)`
would locate backward over two logical blocks.

For most kinds of tape drives, writing a block has the side effect of logically
erasing everything beyond the position of the write. In practice, this side effect
means that most tape drives are append-only devices, because updating a
block in the middle of the tape also effectively erases everything beyond that
block. The tape drive implements this erasing by placing an end-of-tape (EOT)
mark after a block that is written. The drive refuses to locate past the EOT
mark, but it is possible to locate to the EOT and then start writing. Doing so
overwrites the old EOT mark and places a new one at the end of the new blocks
just written.

In principle, a file system can be implemented on a tape. But many of the
file-system data structures and algorithms would be different from those used
for disks, because of the append-only property of tape.

12.9.2.2 File Naming

Another question that the operating system needs to handle is how to name
files on removable media. For a fixed disk, naming is not difficult. On a PC, the
file name consists of a drive letter followed by a path name. In UNIX, the file
name does not contain a drive letter, but the mount table enables the operating
system to discover on what drive the file is located. If the disk is removable,
however, knowing what drive contained the cartridge at some time in the past
does not mean knowing how to find the file. If every removable cartridge in
the world had a different serial number, the name of a file on a removable
device could be prefixed with the serial number, but to ensure that no two
serial numbers are the same would require each one to be about 12 digits in
length. Who could remember the names of her files if she had to memorize a
12-digit serial number for each one?

The problem becomes even more difficult when we want to write data
on a removable cartridge on one computer and then use the cartridge in
another computer. If both machines are of the same type and have the same
kind of removable drive, the only difficulty is knowing the contents and data
layout on the cartridge. But if the machines or drives are different, many
additional problems can arise. Even if the drives are compatible, different
modes. We also need to update the information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

Tertiary storage is built from disk and tape drives that use removable media. Many different technologies are available, including magnetic tape, removable magnetic and magneto-optic disks, and optical disks.

For removable disks, the operating system generally provides the full services of a file-system interface, including space management and request-queue scheduling. For many operating systems, the name of a file on a removable cartridge is a combination of a drive name and a file name within that drive. This convention is simpler but potentially more confusing than is using a name that identifies a specific cartridge.

For tapes, the operating system generally just provides a raw interface. Many operating systems have no built-in support for jukeboxes. Jukebox support can be provided by a device driver or by a privileged application designed for backups or for HSM.

Three important aspects of performance are bandwidth, latency, and reliability. Many bandwidths are available for both disks and tapes. Yet, the random-access latency for a tape is generally much greater than that for a disk. Switching cartridges in a jukebox is also relatively slow. Because a jukebox has a low ratio of drives to cartridges, reading a large fraction of the data in a jukebox can take a long time. Optical media, which protect the sensitive layer with a transparent coating, are generally more robust than magnetic media, which are more likely to expose the magnetic material to physical damage.

Exercises

12.1 None of the disk-scheduling disciplines, except FCFS, is truly fair (starvation may occur).

   a. Explain why this assertion is true.

   b. Describe a way to modify algorithms such as SCAN to ensure fairness.

   c. Explain why fairness is an important goal in a time-sharing system.

   d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.

12.2 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is:

   86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?
Controllers. If you look at a disk drive, you will see a circuit board attached to one side. This is the disk controller. It implements the disk side of the protocol for some kind of connection—SCSI or ATA, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

How can the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support memory-mapped I/O. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.

Some systems use both techniques. For instance, PCs use I/O instructions to control some devices and memory-mapped I/O to control others. Figure 13.2 shows the usual I/O port addresses for PCs. The graphics controller has I/O ports for basic control operations, but the controller has a large memory-mapped region to hold screen contents. The process sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. But the ease of writing...
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 by the interrupt-controller hardware.

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.

The interrupt mechanism accepts an address—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the interrupt vector. This vector contains the memory addresses of specialized interrupt handlers. 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 the technique of interrupt chaining, in which each element in the 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 the one 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 13.4 illustrates the design of the interrupt vector for the Intel Pentium processor. 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. This mechanism enables the CPU to defer the handling of low-priority interrupts without masking off all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of exceptions, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: They are occurrences that induce the CPU to execute an urgent, self-contained routine.

An operating system has other good uses for an efficient hardware and software mechanism that saves a small amount of processor state and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state...
ticks are used to maintain the system time-of-day clock, the system clock can drift. In most computers, the hardware clock is constructed from a high-frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered a high-resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

### 13.3.4 Blocking and Nonblocking I/O

Another aspect of the system-call interface relates to the choice between blocking I/O and nonblocking I/O. When an application issues a blocking system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, most operating systems use blocking system calls at the application interface, because blocking application code is easier to understand than nonblocking application code.

Some user-level processes need nonblocking I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the screen.

One way an application writer can overlap execution with I/O is to write a multithreaded application. Some threads can perform blocking system calls, while others continue executing. The Solaris developers used this technique to implement a user-level library for asynchronous I/O, freeing the application writer from that task. Some operating systems provide nonblocking I/O system calls. A nonblocking call does not halt the execution of the application for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The application continues to execute its code. The completion of the I/O at some future time is communicated to the application, either through the setting of some variable in the address space of the application or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the application. The difference between nonblocking and asynchronous system calls is that a nonblocking read() returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous read() call requests a transfer that will be performed in its entirety but that will complete at some future time. These two I/O methods are shown in Figure 13.8.

A good example of nonblocking behavior is the select() system call for network sockets. This system call takes an argument that specifies a maximum waiting time. By setting it to 0, an application can poll for network activity without blocking. But using select() introduces extra overhead, because the select() call only checks whether I/O is possible. For a data transfer,
1. A process issues a blocking read() system call to a file descriptor of a file that has been opened previously.

2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.

3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.

4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.

5. The device controller operates the device hardware to perform the data transfer.

6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.

7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.

8. The device driver receives the signal, determines which I/O request has completed, determines the request’s status, and signals the kernel I/O subsystem that the request has been completed.

9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.

10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

### 13.6 STREAMS

UNIX System V has an interesting mechanism, called STREAMS, that enables an application to assemble pipelines of driver code dynamically. A stream is a full-duplex connection between a device driver and a user-level process. It consists of a stream head that interfaces with the user process, a driver end that controls the device, and zero or more stream modules between them. The stream head, the driver end, and each module contain a pair of queues—a read queue and a write queue. Message passing is used to transfer data between queues. The STREAMS structure is shown in Figure 13.14.

Modules provide the functionality of STREAMS processing; they are pushed onto a stream by use of the ioctl() system call. For example, a process can
mainframes and in other high-end systems. The job of a channel is to offload I/O work from the main CPU. The idea is that the channels keep the data flowing smoothly, while the main CPU remains free to process the data. Like the device controllers and DMA controllers found in smaller computers, a channel can process more general and sophisticated programs, so channels can be tuned for particular workloads.

We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.
- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).
- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

Devices vary greatly in complexity. For instance, a mouse is simple. The mouse movements and button clicks are converted into numeric values that are fed from hardware through the mouse device driver, to the application. By contrast, the functionality provided by the Windows NT disk device driver is complex. It not only manages individual disks but also implements RAID arrays (Section 12.7). To do so, it converts an application's read or write request into a coordinated set of disk I/O operations. Moreover, it implements sophisticated error-handling and data-recovery algorithms and takes many steps to optimize disk performance.

Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 13.16.

- Initially, we implement experimental I/O algorithms at the application level, because application code is flexible and application bugs are unlikely to cause system crashes. Furthermore, by developing code at the application level, we avoid the need to reboot or reload device drivers after every change to the code. An application-level implementation can be inefficient, however, because of the overhead of context switches and because the application cannot take advantage of internal kernel data structures and kernel functionality (such as efficient in-kernel messaging, threading, and locking).

- When an application-level algorithm has demonstrated its worth, we may reimplement it in the kernel. This can improve the performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system.
13.7 Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces, one of which executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

13.8 Some DMA controllers support direct virtual memory access, where the targets of I/O operations are specified as virtual addresses and a translation from virtual to physical address is performed during the DMA. How does this design complicate the design of the DMA controller? What are the advantages of providing such a functionality?

13.9 UNIX coordinates the activities of the kernel I/O components by manipulating shared in-kernel data structures, whereas Windows NT uses object-oriented message passing between kernel I/O components. Discuss three pros and three cons of each approach.

13.10 Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.

13.11 Discuss the advantages and disadvantages of guaranteeing reliable transfer of data between modules in the STREAMS abstraction.

Bibliographical Notes


The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined indeed hold. More specifically, we must ensure that a process executing in domain $D_i$ can access only those objects specified in row $\backslash$ and then only as allowed by the access-matrix entries.

The access matrix can implement policy decisions concerning protection. The policy decisions involve which rights should be included in the $(i,j)$th entry. We must also decide the domain in which each process executes. This last policy is usually decided by the operating system.

The users normally decide the contents of the access-matrix entries. When a user creates a new object $O_j$, the column $O_j$ is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column / and other rights in other entries, as needed.

The access matrix provides an appropriate mechanism for defining and implementing strict control for both the static and dynamic association between processes and domains. When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix. Similarly, when we change the content of the access matrix, we are performing an operation on an object: the access matrix. Again, we can control these changes by including the access matrix itself as an object. Actually, since each entry in the access matrix may be modified individually, we must consider each entry in the access matrix as an object to be protected. Now, we need to consider only the operations possible on these new objects (domains and the access matrix) and decide how we want processes to be able to execute these operations.

Processes should be able to switch from one domain to another. Domain switching from domain $D_i$ to domain $D_j$ is allowed if and only if the access right switch e access$(i,j)$. Thus, in Figure 14.4, a process executing in domain $D_2$ can switch to domain $D_3$ or to domain $D_4$. A process in domain $D_4$ can switch to $D_1$, and one in domain $D_4$ can switch to domain $D_2$.

![Figure 14.4 Access matrix of Figure 14.3 with domains as objects.](image)
Notice that this facility is similar to the access matrix described in Section 14.4. This relationship will be further explored in the exercises at the end of the chapter.

14.7 Revocation of Access Rights

In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users. Various questions about revocation may arise:

- Immediate versus delayed. Does revocation occur immediately or is it delayed? If revocation is delayed, can we find out when it will take place?
- Selective versus general. When an access right to an object is revoked, does it affect all the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
- Partial versus total. Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- Temporary versus permanent. Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

With an access-list scheme, revocation is easy. The access list is searched for any access rights to be revoked, and they are deleted from the list. Revocation is immediate and can be general or selective, total or partial, and permanent or temporary.

Capabilities, however, present a much more difficult revocation problem. Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

- Reacquisition. Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
- Back-pointers. A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.
- Indirection. The capabilities point indirectly, not directly, to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object. The object for a
capability and its table entry must match. This scheme was adopted in the CAL system. It does not allow selective revocation.

- **Keys.** A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning the capability. A master key is associated with each object; it can be defined or replaced with the `set-key` operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised. Revocation replaces the master key with a new value via the `set-key` operation, invalidating all previous capabilities for this object.

  This scheme does not allow selective revocation, since only one master key is associated with each object. If we associate a list of keys with each object, then selective revocation can be implemented. Finally, we can group all keys into one global table of keys. A capability is valid only if its key matches some key in the global table. We implement revocation by removing the matching key from the table. With this scheme, a key can be associated with several objects, and several keys can be associated with each object, providing maximum flexibility.

  In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users. In particular, it would be reasonable to allow only the owner of an object to set the keys for that object. This choice, however, is a policy decision that the protection system can implement but should not define.

### 14.8 Capability-Based Systems

In this section, we survey two capability-based protection systems. These systems vary in their complexity and in the types of policies that can be implemented on them. Neither system is widely used, but they are interesting proving grounds for protection theories.

#### 14.8.1 An Example: Hydra

Hydra is a capability-based protection system that provides considerable flexibility. A fixed set of possible access rights is known to and interpreted by the system. These rights include such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights. The interpretation of user-defined rights is performed solely by the user's program, but the system provides access protection for the use of these rights, as well as for the use of system-defined rights. These facilities constitute a significant development in protection technology.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with objects of the user-defined type. When the definition of an object is made known to Hydra, the
1. Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an operating system.

2. Protection requirements can be stated independently of the facilities provided by a particular operating system.

3. The means for enforcement need not be provided by the designer of a subsystem.

4. A declarative notation is natural because access privileges are closely related to the linguistic concept of data type.

A variety of techniques can be provided by a programming-language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system. For example, suppose a language is used to generate code to run on the Cambridge CAP system. On this system, every storage reference made on the underlying hardware occurs indirectly through a capability. This restriction prevents any process from accessing a resource outside of its protection environment at any time. However, a program may impose arbitrary restrictions on how a resource can be used during execution of a particular code segment. We can implement such restrictions more rigidly by using the software capabilities provided by CAP. A language implementation might provide standard protected procedures to interpret software capabilities that would realize the protection policies that could be specified in the language. This scheme puts policy specification at the disposal of the programmers, while freeing them from implementing its enforcement.

Even if a system does not provide a protection kernel as powerful as those of Hydra or CAP, mechanisms are still available for implementing protection specifications given in a programming language. The principal distinction is that the security of this protection will not be as great as that supported by a protection kernel, because the mechanism must rely on more assumptions about the operational state of the system. A compiler can separate references for which it can certify that no protection violation could occur from those for which a violation might be possible, and it can treat them differently. The security provided by this form of protection rests on the assumption that the code generated by the compiler will not be modified prior to or during its execution.

What, then, are the relative merits of enforcement based solely on a kernel, as opposed to enforcement provided largely by a compiler?

• Security. Enforcement by a kernel provides a greater degree of security of the protection system itself than does the generation of protection-checking code by a compiler. In a compiler-supported scheme, security rests on correctness of the translator, on some underlying mechanism of storage management that protects the segments from which compiled code is executed, and, ultimately, on the security of files from which a program is loaded. Some of these considerations also apply to a software-supported protection kernel, but to a lesser degree, since the kernel may reside in fixed physical storage segments and may be loaded from only a designated file. With a tagged-capability system, in which all address
computation is performed either by hardware or by a fixed microprogram, even greater security is possible. Hardware-supported protection is also relatively immune to protection violations that might occur as a result of either hardware or system software malfunction.

- Flexibility. There are limits to the flexibility of a protection kernel in implementing a user-defined policy, although it may supply adequate facilities for the system to provide enforcement of its own policies. With a programming language, protection policy can be declared and enforcement provided as needed by an implementation. If a language does not provide sufficient flexibility, it can be extended or replaced with less disturbance of a system in service than would be caused by the modification of an operating-system kernel.

- Efficiency. The greatest efficiency is obtained when enforcement of protection is supported directly by hardware (or microcode). Insofar as software support is required, language-based enforcement has the advantage that static access enforcement can be verified off-line at compile time. Also, since an intelligent compiler can tailor the enforcement mechanism to meet the specified need, the fixed overhead of kernel calls can often be avoided.

In summary, the specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources. A language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable. In addition, it can interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

One way of making protection available to the application program is through the use of a software capability that could be used as an object of computation. Inherent in this concept is the idea that certain program components might have the privilege of creating or examining these software capabilities. A capability-creating program would be able to execute a primitive operation that would seal a data structure, rendering the latter’s contents inaccessible to any program components that did not hold either the seal or the unseal privilege. They might copy the data structure or pass its address to other program components, but they could not gain access to its contents. The reason for introducing such software capabilities is to bring a protection mechanism into the programming language. The only problem with the concept as proposed is that the use of the seal and unseal operations takes a procedural approach to specifying protection. A nonprocedural or declarative notation seems a preferable way to make protection available to the application programmer.

What is needed is a safe, dynamic access-control mechanism for distributing capabilities to system resources among user processes. To contribute to the overall reliability of a system, the access-control mechanism should be safe to use. To be useful in practice, it should also be reasonably efficient. This requirement has led to the development of a number of language constructs that allow the programmer to declare various restrictions on the use of a specific managed resource. (See the Bibliographical Notes for appropriate references.) These constructs provide mechanisms for three functions:
Protection, as we discussed in Chapter 14, is strictly an internal problem: How do we provide controlled access to programs and data stored in a computer system? Security, on the other hand, requires not only an adequate protection system but also consideration of the external environment within which the system operates. A protection system is ineffective if user authentication is compromised or a program is run by an unauthorized user.

Computer resources must be guarded against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. These resources include information stored in the system (both data and code), as well as the CPU, memory, disks, tapes and networking that are the computer. In this chapter, we start by examining ways in which resources may be accidentally or purposefully misused. We then explore a key security enabler—cryptography. Finally, we look at mechanisms to guard against or detect attacks.

CHAPTER OBJECTIVES

- To discuss security threats and attacks.
- To explain the fundamentals of encryption, authentication, and hashing.
- To examine the uses of cryptography in computing.
- To describe the various countermeasures to security attacks.

15.1 The Security Problem

In many applications, ensuring the security of the computer system is worth considerable effort. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether by accident or fraud, can seriously impair the ability of the corporation to function.

In Chapter 14, we discussed mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect
their resources, including programs and data. These mechanisms work well only as long as the users conform to the intended use of and access to these resources. We say that a system is **secure** if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Nonetheless, we must have mechanisms to make security breaches a rare occurrence, rather than the norm.

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of protection from accidents. The following list includes forms of accidental and malicious security violations. We should note that in our discussion of security, we use the terms **intruder** and **cracker** for those attempting to breach security. In addition, a **threat** is the potential for a security violation, such as the discovery of a vulnerability, whereas an **attack** is the attempt to break security.

- **Breach of confidentiality.** This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, can result directly in money for the intruder.

- **Breach of integrity.** This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial application.

- **Breach of availability.** This violation involves unauthorized destruction of data. Some crackers would rather wreak havoc and gain status or bragging rights than gain financially. Web-site defacement is a common example of this type of security breach.

- **Theft of service.** This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.

- **Denial of service.** This violation involves preventing legitimate use of the system. **Denial-of-service** or **DOS**, attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread. We discuss DOS attacks further in Section 15.3.3.

Attackers use several standard methods in their attempts to breach security. The most common is **masquerading**, in which one participant in a communication pretends to be someone else (another host or another person). By masquerading, attackers breach **authentication**, the correctness of identification; they can then can gain access that they would not normally be allowed or escalate their privileges—obtain privileges to which they would not normally be entitled. Another common attack is to replay a captured exchange of data. A **replay attack** consists of the malicious or fraudulent repeat of a valid data transmission. Sometimes the replay comprises the entire attack—for example, in a repeat of a request to transfer money. But frequently it is done along with **message modification**, again to escalate privileges. Consider the damage that could be done if a request for authentication had a legitimate
To protect a system, we must take security measures at four levels:

1. **Physical.** The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders. Both the machine rooms and the terminals or workstations that have access to the machines must be secured.

2. **Human.** Authorizing users must be done carefully to assure that only appropriate users have access to the system. Even authorized users, however, may be "encouraged" to let others use their access (in exchange for a bribe, for example). They may also be tricked into allowing access via social engineering. One type of social-engineering attack is phishing. Here, a legitimate-looking e-mail or web page misleads a user into entering confidential information. Another technique is dumpster diving, a general term for attempting to gather information in order to gain unauthorized access to the computer (by looking through trash, finding phone books, or finding notes containing passwords, for example). These security problems are management and personnel issues, not problems pertaining to operating systems.

3. **Operating system.** The system must protect itself from accidental or purposeful security breaches. A runaway process could constitute an accidental denial-of-service attack. A query of a service could reveal passwords. A stack overflow could allow the launching of an unauthorized process. The list of possible breaches is almost endless.

4. **Network.** Much computer data in modern systems travels over private or shared lines like the Internet, wireless connections, or dial-up lines. Intercepting these data could be just as harmful as breaking into a computer; and interruption of communications could constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.

Security at the first two levels must be maintained if operating-system security is to be ensured. A weakness at a high level of security (physical or human) allows circumvention of strict low-level (operating-system) security measures. Thus, the old adage that a chain is as weak as its weakest link is especially true of system security. All of these aspects must be addressed for security to be maintained.

Furthermore, the system must provide protection (Chapter 14) to allow the implementation of security features. Without the ability to authorize users and processes, to control their access, and to log their activities, it would be impossible for an operating system to implement security measures or to run securely. Hardware protection features are needed to support an overall protection scheme. For example, a system without memory protection cannot be secure. New hardware features are allowing systems to be made more secure, as we shall discuss.

Unfortunately, little in security is straightforward. As intruders exploit security vulnerabilities, security countermeasures are created and deployed. This causes intruders to become more sophisticated in their attacks. For example, recent security incidents include the use of spyware to provide a conduit for spam through innocent systems (we discuss this practice in...
• **Tunneling.** This virus attempts to bypass detection by an antivirus scanner by installing itself in the interrupt-handler chain. Similar viruses install themselves in device drivers.

• **Multipartite.** A virus of this type is able to infect multiple parts of a system, including boot sectors, memory, and files. This makes it difficult to detect and contain.

• **Armored.** An armored virus is coded to make itself hard for antivirus researchers to unravel and understand. It can also be compressed to avoid detection and disinfection. In addition, virus droppers and other full files that are part of a virus infestation are frequently hidden via file attributes or unviewable file names.

This vast variety of viruses is likely to continue to grow. In fact, in 2004 a new and widespread virus was detected. It exploited three separate bugs for its operation. This virus started by infecting hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server (IIS). Any vulnerable Microsoft Explorer web browser visiting those sites received a browser virus with any download. The browser virus installed several back-door programs, including a keystroke logger, which records all things entered on the keyboard (including passwords and credit-card numbers). It also installed a script to allow unlimited remote access by an intruder and another that allowed an intruder to route spam through the infected desktop computer.

In summary, viruses are the most disruptive security attack; and because they are effective, they will continue to be written and to spread. Among the active debates within the computing community is whether a **monoculture**, in which many systems run the same hardware, operating system, and/or application software, is increasing the threat of and damage caused by security intrusions. Within the debate is the issue of whether or not there even exists a monoculture today (consisting of Microsoft products).

### 15.3 System and Network Threats

Program threats typically use a breakdown in the protection mechanisms of a system to attack programs. In contrast, system and network threats involve the abuse of services and network connections. Sometimes a system and network attack is used to launch a program attack, and vice versa.

System and network threats create a situation in which operating-system resources and user files are misused. Here, we discuss some examples of these threats, including worms, port scanning, and denial-of-service attacks.

It is important to note that masquerading and replay attacks are also common over networks between systems. In fact, these attacks are more effective and harder to counter when multiple systems are involved. For example, within a computer, the operating system usually can determine the sender and receiver of a message. Even if the sender changes to the ID of someone else, there might be a record of that ID change. When multiple systems are involved, especially systems controlled by attackers, then such tracing is much harder.
The generalization is that sharing secrets (to prove identity and as keys to encryption) is required for authentication and encryption, and that is easier in environments (such as a single operating system) in which secure sharing methods exist. These methods include shared memory and interprocess communications. Creating secure communication and authentication is discussed in Sections 15.4 and 15.5.

15.3.1 Worms

A worm is a process that uses the spawn mechanism to ravage system performance. The worm spawns copies of itself, using up system resources and perhaps locking out all other processes. On computer networks, worms are particularly potent, since they may reproduce themselves among systems and thus shut down an entire network. Such an event occurred in 1988 to UNIX systems on the Internet, causing millions of dollars of lost system and system administrator time.

At the close of the workday on November 2, 1988, Robert Tappan Morris, Jr., a first-year Cornell graduate student, unleashed a worm program on one or more hosts connected to the Internet. Targeting Sun Microsystems' Sun 3 workstations and VAX computers running variants of Version 4 BSD UNIX, the worm quickly spread over great distances; within a few hours of its release, it had consumed system resources to the point of bringing down the infected machines.

Although Robert Morris designed the self-replicating program for rapid reproduction and distribution, some of the features of the UNIX networking environment provided the means to propagate the worm throughout the system. It is likely that Morris chose for initial infection an Internet host left open for and accessible to outside users. From there, the worm program exploited flaws in the UNIX operating system's security routines and took advantage of UNIX utilities that simplify resource sharing in local-area networks to gain unauthorized access to thousands of other connected sites. Morris's methods of attack are outlined next.

![Figure 15.6 The Morris Internet worm.](image-url)
With each new access, the worm program searched for already active copies of itself. If it found one, the new copy exited, except in every seventh instance. Had the worm exited on all duplicate sightings, it might have remained undetected. Allowing every seventh duplicate to proceed (possibly to confound efforts to stop its spread by baiting with fake worms) created a wholesale infestation of Sun and VAX systems on the Internet.

The very features of the UNIX network environment that assisted the worm's propagation also helped to stop its advance. Ease of electronic communication, mechanisms to copy source and binary files to remote machines, and access to both source code and human expertise allowed cooperative efforts to develop solutions quickly. By the evening of the next day, November 3, methods of halting the invading program were circulated to system administrators via the Internet. Within days, specific software patches for the exploited security flaws were available.

Why did Morris unleash the worm? The action has been characterized as both a harmless prank gone awry and a serious criminal offense. Based on the complexity of starting the attack, it is unlikely that the worm's release or the scope of its spread was unintentional. The worm program took elaborate steps to cover its tracks and to repel efforts to stop its spread; the program contained no code aimed at damaging or destroying systems on which it ran. The author clearly had the expertise to include such commands; in fact, data structures were present in the code that could have been used to transfer Trojan-horse or virus programs. The behavior of the program may lead to interesting observations, but it does not provide a sound basis for inferring motive. Why is it open to speculation, however, is the legal outcome: A federal court convicted Morris and handed down a sentence of three years' probation, 400 hours of community service, and a $10,000 fine. Morris's legal costs probably exceeded $100,000.

Security experts continue to evaluate methods to decrease or eliminate worms. A more recent event, though, shows that worms are still a fact of life on the Internet. It also shows that as the Internet grows, the damage that even "harmless" worms can do also grows and can be significant. This example occurred during August 2003. The fifth version of the "Sobig" worm, more properly known as "W32.Sobig.F@mm," was released by persons at this time unknown. It was the fastest-spreading worm released to date, at its peak infecting hundreds of thousands of computers and one in seventeen e-mail messages on the Internet. It clogged e-mail inboxes, slowed networks, and took a huge number of hours to clean up.

Sobig.F was launched by being uploaded to a pornography newsgroup via an account created with a stolen credit card. It was disguised as a photo. The virus targeted Microsoft Windows systems and used its own SMTP engine to e-mail itself to all the addresses found on an infected system. It used a variety of subject lines to help avoid detection, including "Thank You!" "Your details," and "Re: Approved." It also used a random address on the host as the "From:" address, making it difficult to determine from the message which machine was the infected source. Sobig.F included an attachment for the target e-mail reader to click on, again with a variety of names. If this payload was executed, it stored a program called W1NPPR32.EXE in the default Windows directory, along with a text file. It also modified the Windows registry.
situation is quite different. A networked computer receives bits *from the wire* with no immediate and reliable way of determining what machine or application sent those bits. Similarly, the computer sends bits onto the network with no way of knowing who might eventually receive them.

Commonly, network addresses are used to infer the potential senders and receivers of network messages. Network packets arrive with a source address, such as an IP address. And when a computer sends a message, it names the intended receiver by specifying a destination address. However, for applications where security matters, we are asking for trouble if we assume that the source or destination address of a packet reliably determines who sent or received that packet. A rogue computer can send a message with a falsified source address, and numerous computers other than the one specified by the destination address can (and typically do) receive a packet. For example, all of the routers on the way to the destination will receive the packet, too. How, then, is an operating system to decide whether to grant a request when it cannot trust the named source of the request? And how is it supposed to provide protection for a request or data when it cannot determine who will receive the response or message contents it sends over the network?

It is generally considered infeasible to build a network of any scale in which the source and destination addresses of packets can be trusted in this sense. Therefore, the only alternative is somehow to eliminate the need to trust the network. This is the job of cryptography. Abstractly, cryptography is used to constrain the potential senders and/or receivers of a message. Modern cryptography relies on secrets called keys that are selectively distributed to computers in a network and used to process messages. Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key—the key is the source of the message. Similarly, a sender can encode its message so that only a computer with a certain key can decode the message, so that the key becomes the destination. Unlike network addresses, however, keys are designed so that it is not computationally feasible to derive them from the messages they were used to generate or from any other public information. Thus, they provide a much more trustworthy means of constraining senders and receivers of messages. Note that cryptography is a field of study unto itself, with large and small complexities and subtleties. Here, we explore the most important aspects of the parts of cryptography that pertain to operating systems.

### 15.4.1 Encryption

Because it solves a wide variety of communication security problems, encryption is used frequently in many aspects of modern computing. Encryption is a means for constraining the possible receivers of a message. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message. Encryption of messages is an ancient practice, of course, and there have been many encryption algorithms, dating back to before Caesar. In this section, we describe important modern encryption principles and algorithms.

Figure 15.7 shows an example of two users communicating securely over an insecure channel. We refer to this figure throughout the section. Note that the
A secure communication over an insecure medium.

Figure 15.7

Key exchange can take place directly between the two parties or via a trusted third party (that is, a certificate authority), as discussed in Section 15.4.1.4.

An encryption algorithm consists of the following components:

• A set $K$ of keys.
• A set $M$ of messages.
• A set $C$ of ciphertexts.
• A function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, $E(k)$ is a function for generating ciphertexts from messages. Both $E$ and $E(k)$ for any $k$ should be efficiently computable functions.
• A function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, $D(k)$ is a function for generating messages from ciphertexts. Both $D$ and $D(k)$ for any $k$ should be efficiently computable functions.

An encryption algorithm must provide this essential property: Given a ciphertext $c \in C$, a computer can compute $m$ such that $E(k)(m) = c$ only if it possesses $D(k)$. Thus, a computer holding $D(k)$ can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding $D(k)$ cannot decrypt ciphertexts. Since ciphertexts are generally exposed (for example, sent...
To send a message $m$ to the server, the client sends

$$c = E(k_{cs}^{\text{crypt}})(\langle m, S(k_{sc}^{\text{mac}})(m) \rangle).$$

Upon receiving $c$, the server recovers

$$\langle m, a \rangle = D(k_{cs}^{\text{crypt}})(c)$$

and accepts $m$ if $V(k_{sc}^{\text{mac}})(m, a) = \text{true}$. Similarly, to send a message $m$ to the client, the server sends

$$c = E(k_{sc}^{\text{crypt}})(\langle m, S(k_{sc}^{\text{mac}})(m) \rangle)$$

and the client recovers

$$\langle m, a \rangle = D(k_{sc}^{\text{crypt}})(c)$$

and accepts $m$ if $V(k_{sc}^{\text{mac}})(m, a) = \text{true}$.

This protocol enables the server to limit the recipients of its messages to the client that generated $\text{prns}$ and to limit the senders of the messages it accepts to that same client. Similarly, the server can limit the recipients of the messages it sends and the senders of the messages it accepts to the party that knows $S(k_d)$ (that is, the party that can decrypt $\text{pm}$). In many applications, such as web transactions, the client needs to identify the identity of the party that knows $S(k_d)$. This is one purpose of the certificate $\text{cert}$; in particular, the $\text{attrs}$ field contains information that the client can use to determine the identity—for example, the domain name—of the server with which it is communicating. For applications in which the server also needs information about the client, SSL supports an option by which a client can send a certificate to the server.

In addition to its use on the Internet, SSL is being used for a wide variety of tasks. For example, IPSec VPNs now have a competitor in SSL VPNs. IPSec is good for point-to-point encryption of traffic—say, between two company offices. SSL VPNs are more flexible but not as efficient, so they might be used between an individual employee working remotely and the corporate office.

15.5 User Authentication

The discussion of authentication above involves messages and sessions. But what of users? If a system cannot authenticate a user, then authenticating that a message came from that user is pointless. Thus, a major security problem for operating systems is user authentication. The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system. A user normally identifies herself. How do we determine whether a user’s identity is authentic? Generally, user authentication is based on one or more of three things: the user’s possession of something (a key or card), the user’s knowledge of something (a user identifier and password), and/or an attribute of the user (fingerprint, retina pattern, or signature).
could be copied for off-system analysis. Or consider a Trojan-horse program installed on the system that captures every keystroke before sending it on to the application.

Exposure is a particularly severe problem if the password is written down where it can be read or lost. As we shall see, some systems force users to select hard-to-remember or long passwords, which may cause a user to record the password or to reuse it. As a result, such systems provide much less security than systems that allow users to select easy passwords!

The final type of password compromise, illegal transfer, is the result of human nature. Most computer installations have a rule that forbids users to share accounts. This rule is sometimes implemented for accounting reasons but is often aimed at improving security. For instance, suppose one user ID is shared by several users, and a security breach occurs from that user ID. It is impossible to know who was using the ID at the time the break occurred or even whether the user was an authorized one. With one user per user ID, any user can be questioned directly about use of the account; in addition, the user might notice something different about the account and detect the break-in. Sometimes, users break account-sharing rules to help friends or to circumvent accounting, and this behavior can result in a system's being accessed by unauthorized users—possibly harmful ones.

Passwords can be either generated by the system or selected by a user. System-generated passwords may be difficult to remember, and thus users may write them down. User-selected passwords are often easy to guess (the user's name or a favorite car, for example). Some systems will check a proposed password for ease of guessing or cracking before accepting it. At some sites, administrators occasionally check user passwords and notify a user if his password is easy to guess. Some systems also age passwords, forcing users to change their passwords at regular intervals (every three months, for instance). This method is not foolproof either, because users can easily toggle between two passwords. The solution, as implemented on some systems, is to record a password history for each user. For instance, the system could record the last $N$ passwords and not allow their reuse.

Several variants on these simple password schemes can be used. For example, the password can be changed more frequently. In the extreme, the password is changed from session to session. A new password is selected (either by the system or by the user) at the end of each session, and that password must be used for the next session. In such a case, even if a password is misused, it can be used only once. When the legitimate user tries to use a now-invalid password at the next session, he discovers the security violation. Steps can then be taken to repair the breached security.

### 15.5.3 Encrypted Passwords

One problem with all these approaches is the difficulty of keeping the password secret within the computer. How can the system store a password securely yet allow its use for authentication when the user presents her password? The UNIX system uses encryption to avoid the necessity of keeping its password list secret. Each user has a password. The system contains a function that is extremely difficult—the designers hope impossible—to invert but is simple to compute. That is, given a value $x$, it is easy to compute the function value
Part Six

Distributed Systems

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through communication lines such as local-area or wide-area networks. The processors in a distributed system vary in size and function. Such systems may include small handheld or real-time devices, personal computers, workstations, and large mainframe computer systems.

A distributed file system is a file-service system whose users, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple independent storage devices.

The benefits of a distributed system include giving users access to the resources maintained by the system and thereby speeding up computation and improving data availability and reliability. Because a system is distributed, however, it must provide mechanisms for process synchronization and communication, for dealing with the deadlock problem, and for handling failures that are not encountered in a centralized system.
16.2.2.3 Process Migration

A logical extension of computation migration is process migration. When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:

- **Load balancing.** The processes (or subprocesses) may be distributed across the network to even the workload.

- **Computation speedup.** If a single process can be divided into a number of subprocesses that can run concurrently on different sites, then the total process turnaround time can be reduced.

- **Hardware preference.** The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on an array processor, rather than on a microprocessor).

- **Software preference.** The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.

- **Data access.** Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely than to transfer all the data.

We use two complementary techniques to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. This scheme has the advantage that the user does not need to code her program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely.

The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed when the process must be moved to satisfy a hardware or software preference.

You have probably realized that the Web has many aspects of a distributed-computing environment. Certainly it provides data migration (between a web server and a web client). It also provides computation migration. For instance, a web client could trigger a database operation on a web server. Finally, with Java, it provides a form of process migration: Java applets are sent from the server to the client, where they are executed. A network operating system provides most of these features, but a distributed operating system makes them seamless and easily accessible. The result is a powerful and easy-to-use facility—one of the reasons for the huge growth of the World Wide Web.

16.3 Network Structure

There are basically two types of networks: **local-area networks (LAN)** and **wide-area networks (WAN)**. The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed
of processors distributed over small areas (such as a single building or a number of adjacent buildings), whereas wide-area networks are composed of a number of autonomous processors distributed over a large area (such as the United States). These differences imply major variations in the speed and reliability of the communications network, and they are reflected in the distributed operating-system design.

16.3.1 Local-Area Networks

Local-area networks emerged in the early 1970s as a substitute for large mainframe computer systems. For many enterprises, it is more economical to have a number of small computers, each with its own self-contained applications, than to have a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs, as mentioned, are usually designed to cover a small geographical area (such as a single building or a few adjacent buildings) and are generally used in an office environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than do their counterparts in wide-area networks. High-quality (expensive) cables are needed to attain this higher speed and reliability. It is also possible to use the cable exclusively for data network traffic. Over longer distances, the cost of using high-quality cable is enormous, and the exclusive use of the cable tends to be prohibitive.

![Figure 16.2 Local-area network.](image-url)
The most common links in a local-area network are twisted-pair and fiber-optic cabling. The most common configurations are multiaccess bus, ring, and star networks. Communication speeds range from 1 megabit per second, for networks such as AppleTalk, infrared, and the new Bluetooth local radio network, to 1 gigabit per second for gigabit Ethernet. Ten megabits per second is most common and is the speed of 10BaseT Ethernet. 100BaseT Ethernet requires a higher-quality cable but runs at 100 megabits per second and is becoming common. Also growing is the use of optical-fiber-based FDDI networking. The FDDI network is token-based and runs at over 100 megabits per second.

A typical LAN may consist of a number of different computers (from mainframes to laptops or PDAs), various shared peripheral devices (such as laser printers and magnetic-tape drives), and one or more gateways (specialized processors) that provide access to other networks (Figure 16.2). An Ethernet scheme is commonly used to construct LANs. An Ethernet network has no central controller, because it is a multiaccess bus, so new hosts can be added easily to the network. The Ethernet protocol is defined by the IEEE 802.3 standard.

16.3.2 Wide-Area Networks

Wide-area networks emerged in the late 1960s, mainly as an academic research project to provide efficient communication; not only was this allowing hardware and software to be shared conveniently and economically by a wide community of users, but also the first WAN to be designed and developed was the Arpanet. Begun in 1968, the Arpanet has grown from a four-site experimental network to a worldwide network of networks, the Internet, comprising millions of computer systems.

Because the sites in a WAN are physically distributed over a large geographical area, the communication links are, by default, relatively slow and unreliable. Typical links are telephone lines, leased (dedicated data) lines, microwave links, and satellite channels. These communication links are controlled by special communication processors (Figure 16.3), which are responsible for defining the interface through which the sites communicate over the network, as well as for transferring information among the various sites.

For example, the Internet WAN provides the ability for hosts at geographically separated sites to communicate with one another. The host computers typically differ from one another in type, speed, word length, operating system, and so on. Hosts are generally on LANs, which are, in turn, connected to the Internet via regional networks. The regional networks, such as NSFnet in the northeast United States, are interlinked with routers (Section 16.5.2) to form the worldwide network. Connections between networks frequently use a telephone-system service called T1, which provides a transfer rate of 1.544 megabits per second over a leased line. For sites requiring faster Internet access, T1s are collected into multiple-T1 units that work in parallel to provide more throughput. For instance, a T3 is composed of 28 T1 connections and has a transfer rate of 45 megabits per second. The routers control the path each message takes through the net. This routing may be either dynamic, to increase communication efficiency, or static, to reduce security risks or to allow communication charges to be computed.
of a distributed system is a potential for fault tolerance and scalability because of the multiplicity of resources. However, inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Very large-scale distributed systems, to a great extent, are still only theoretical. No magic guidelines ensure the scalability of a system. It is easier to point out why current designs are not scalable. We next discuss several designs that pose problems and propose possible solutions, all in the context of scalability.

One principle for designing very large-scale systems is that the service demand from any component of the system should be bounded by a constant that is independent of the number of nodes in the system. Any service mechanism whose load demand is proportional to the size of the system is destined to become clogged once the system grows beyond a certain size. Adding more resources will not alleviate such a problem. The capacity of this mechanism simply limits the growth of the system.

Central control schemes and central resources should not be used to build scalable (and fault-tolerant) systems. Examples of central entities are central authentication servers, central naming servers, and central file servers. Centralization is a form of functional asymmetry among machines constituting the system. The ideal alternative is a functionally symmetric configuration; that is, all the component machines have an equal role in the operation of the system, and hence each machine has some degree of autonomy. Practically, it is virtually impossible to comply with such a principle. For instance, incorporating diskless machines violates functional symmetry; since the workstations depend on a central disk. However, autonomy and symmetry are important goals to which we should aspire.

The practical approximation of symmetric and autonomous configuration is clustering, in which the system is partitioned into a collection of semi-autonomous clusters. A cluster consists of a set of machines and a dedicated cluster server. So that cross-cluster resource references are relatively infrequent, each cluster server should satisfy requests of its own machines most of the time. Of course, this scheme depends on the ability to localize resource references and to place the component units appropriately. If the cluster is well balanced—that is, if the server in charge suffices to satisfy all the cluster demands—it can be used as a modular building block to scale up the system.

Deciding on the process structure of the server is a major problem in the design of any service. Servers are supposed to operate efficiently in peak periods, when hundreds of active clients need to be served simultaneously. A single-process server is certainly not a good choice, since whenever a request necessitates disk I/O, the whole service will be blocked. Assigning a process for each client is a better choice; however, the expense of frequent context switches between the processes must be considered. A related problem occurs because all the server processes need to share information.

One of the best solutions for the server architecture is the use of lightweight processes, or threads, which we discussed in Chapter 4. We can think of a group of lightweight processes as multiple threads of control associated with some shared resources. Usually, a lightweight process is not bound to a particular client. Instead, it serves single requests of different clients. Scheduling of threads can be preemptive or nonpreemptive. If threads are allowed to run
that packet's destination? Does a token-passing network need such a protocol? Explain your answer.

16.15 What are the advantages and the disadvantages of making the computer network transparent to the user?

Bibliographical Notes


The Internet and its protocols were described in Comer [1999] and Comer [2000]. Coverage of TCP/IP can be found in Stevens [1994] and Stevens [1995]. UNIX network programming was described thoroughly in Stevens [1997] and Stevens [1998].

Discussions concerning distributed operating-system structures have been offered by Coulouris et al. [2001] and Tanenbaum and van Steen [2002].

Load balancing and load sharing were discussed by Harchol-Balter and Downey [1997] and Vee and Hsu [2000]. Harish and Ghosh [1999] described load-balancing DNS servers. Process migration was discussed by Jul et al. [1988], Douglis and Ousterhout [1991], Han and Ghosh [1998], and Milojicic et al. [2000]. Issues relating to a distributed virtual machine for distributed systems were examined in Sirer et al. [1999].
17.3.1 Basic Caching Scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of those data is brought from the server to the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy (for example, least recently used) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies (or parts) of the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is the cache-consistency problem, which we discuss in Section 17.3.4. DFS caching could just as easily be called network virtual memory; it acts similarly to demand-paged virtual memory, except that the backing store usually is not a local disk but rather a remote server. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, notwithstanding the resulting performance penalty.

The granularity of the cached data in DFS can vary from blocks of a file to an entire file. Usually, more data is cached than are needed to satisfy a single access, so that many accesses can be served by the cached data. This procedure is much like disk read-ahead (Section 11.6.2). AFS caches files in large chunks (64 KB). The other systems discussed in this chapter support caching of individual blocks driven by client demand; increasing the caching unit increases the hit ratio, but it also increases the miss penalty, because each miss requires more data to be transferred. It increases the potential for consistency problems as well. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (if an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5 KB, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

Block size and total cache size are obviously of importance for block-caching schemes. In UNIX-like systems, common block sizes are 4 KB and 8 KB. For large caches (over 1 MB), large block sizes (over 8 KB) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and a lower hit ratio.

17.3.2 Cache Location

Where should the cached data be stored—on disk or in main memory? Disk caches have one clear advantage over main-memory caches: They are reliable. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, they are still there during recovery, and there is no need to fetch them again. Main-memory caches have several advantages of their own, however:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.
• Technology is moving toward larger and less expensive memory. The achieved performance speedup is predicted to outweigh the advantages of disk caches.

• The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service. In NFS, for instance, the implementation is based on remote service but is augmented with client- and server-side memory caching for performance. Similarly, Sprite's implementation is based on caching; but under certain circumstances, a remote-service method is adopted. Thus, to evaluate the two methods, we must evaluate to what degree either method is emphasized.

The NFS protocol and most implementations do not provide disk caching. Recent Solaris implementations of NFS (Solaris 2.6 and beyond) include a client-side disk caching option, the cachefs file system. Once the NFS client reads blocks of a file from the server, it caches them in memory as well as on disk. If the memory copy is flushed, or even if the client reboots, the disk cache is referenced. If a needed block is neither in memory nor in the cachefs disk cache, an RPC is sent to the server to retrieve the file and the block is written into the disk cache as well as stored in the memory cache for client use.

17.3.3 Cache-Update Policy

The policy used to write modified data blocks back to the server's master copy has a critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as they are placed in any cache. The advantage of a write-through policy is reliability: Little information is lost when a client system crashes. However, this policy requires each write access to wait until the information is sent to the server, so it causes poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses.

An alternative is the delayed-write policy, also known as write-back caching, where we delay updates to the master copy. Modifications are written to the cache and then are written through to the server at a later time. This policy has two advantages over write-through. First, because writes are made to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case only the last update needs to be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data are lost whenever a user machine crashes.

Variations of the delayed-write policy differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server. A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan, just as UNIX scans
files and directories suspect, and it generates a cache-validation request for the first use of each such entry.

The callback mechanism forces each server to maintain callback information and each client to maintain validity information. If the amount of callback information maintained by a server is excessive, the server can break callbacks and reclaim some storage by unilaterally notifying clients and revoking the validity of their cached files. If the callback state maintained by Venus gets out of sync with the corresponding state maintained by the servers, some inconsistency may result.

Venus also caches contents of directories and symbolic links, for pathname translation. Each component in the path name is fetched, and a callback is established for it if it is not already cached or if the client does not have a callback on it. Venus does lookups on the fetched directories locally, using fids. No requests are forwarded from one server to another. At the end of a pathname traversal, all the intermediate directories and the target file are in the cache with callbacks on them. Future open calls to this file will involve no network communication at all, unless a callback is broken on a component of the path name.

The only exception to the caching policy is a modification to a directory that is made directly on the server responsible for the directory for reasons of integrity. The Vice interface has well-defined operations for such purposes. Venus reflects the changes in its cached copy to avoid re-fetching the directory.

17.6.4 Implementation

Client processes are interfaced to a UNIX kernel with the usual set of system calls. The kernel is modified slightly to detect references to Vice files in the relevant operations and to forward the requests to the client-level Venus process at the workstation.

Venus carries out pathname translation component by component, as described above. It has a mapping cache that associates volumes to server locations in order to avoid server interrogation for an already known volume location. If a volume is not present in this cache, Venus contacts any server to which it already has a connection, requests the location information, and enters that information into the mapping cache. Unless Venus already has a connection to the server, it establishes a new connection. It then uses this connection to fetch the file or directory. Connection establishment is needed for authentication and security purposes. When a target file is found and cached, a copy is created on the local disk. Venus then returns to the kernel, which opens the cached copy and returns its handle to the client process.

The UNIX file system is used as a low-level storage system for both AFS servers and clients. The client cache is a local directory on the workstation's disk. Within this directory are files whose names are placeholders for cache entries. Both Venus and server processes access UNIX files directly by the latter's modes to avoid the expensive pathname-to_inode translation routine (namei). Because the internal inode interface is not visible to client-level processes (both Venus and server processes are client-level processes), an appropriate set of additional system calls was added. DFS uses its own journaling file system to improve performance and reliability over UFS.
that has received a reply message from all other processes in the system can enter its critical section, queueing incoming requests and deferring them. After exiting its critical section, the process sends reply messages to all its deferred requests.

The decision whether process \( P_i \) replies immediately to a request\((P_j, TS)\) message or defers its reply is based on three factors:

1. If process \( P_i \) is in its critical section, then it defers its reply to \( P_j \).
2. If process \( P_i \) does not want to enter its critical section, then it sends a reply immediately to \( P_j \).
3. If process \( P_i \) wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp of the incoming request made by process \( P_j \). If its own request timestamp is greater than that of the incoming request, then it sends a reply immediately to \( P_j \) (\( P_j \) asked first). Otherwise, the reply is deferred.

This algorithm exhibits the following desirable behavior:

- Mutual exclusion is obtained.
- Freedom from deadlock is ensured.
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in FCFS order.

The number of messages per critical-section entry is \( 2 \times (n - 1) \). This number is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

To illustrate how the algorithm functions, we consider a system consisting of processes \( P_1, P_2, \) and \( P_3 \). Suppose that processes \( P_1 \) and \( P_3 \) want to enter their critical sections. Process \( P_1 \) then sends a message request \((P_1, \text{timestamp} = 10)\) to processes \( P_2 \) and \( P_3 \), while process \( P_3 \) sends a message request \((P_3, \text{timestamp} = 4)\) to processes \( P_1 \) and \( P_2 \). The timestamps 4 and 10 were obtained from the logical clocks described in Section 18.1. When process \( P_2 \) receives these request messages, it replies immediately. When process \( P_1 \) receives the request from process \( P_3 \), it replies immediately, since the timestamp (10) on its own request message is greater than the timestamp (4) for process \( P_3 \). When process \( P_3 \) receives the request message from process \( P_1 \), it defers its reply, since the timestamp (4) on its request message is less than the timestamp (10) for the message from process \( P_1 \). On receiving replies from both process \( P_2 \) and process \( P_2 \), process \( P_3 \) can enter its critical section. After exiting its critical section, process \( P_3 \) sends a reply to process \( P_1 \), which can then enter its critical section.

Because this scheme requires the participation of all the processes in the system, however, it has three undesirable consequences:

1. The processes need to know the identity of all other processes in the system. When a new process joins the group of processes participating in the mutual-exclusion algorithm, the following actions need to be taken:
• In the wait-die scheme, an older process must wait for a younger one to release its resource. Thus, the older the process gets, the more it tends to wait. By contrast, in the wound-wait scheme, an older process never waits for a younger process.

• In the wait-die scheme, if a process \( P_i \) dies and is rolled back because it has requested a resource held by process \( P_j \), then \( P_i \) may reissue the same sequence of requests when it is restarted. If the resource is still held by \( P_j \), then \( P_i \) will die again. Thus, \( P_i \) may die several times before acquiring the needed resource. Contrast this series of events with what happens in the wound-wait scheme. Process \( P_i \) is wounded and rolled back because \( P_j \) has requested a resource it holds. When \( P_i \) is restarted and requests the resource now being held by \( P_j \), \( P_i \) waits. Thus, fewer rollbacks occur in the wound-wait scheme.

The major problem with both schemes is that unnecessary rollbacks may occur.

18.5.2 Deadlock Detection

The deadlock-prevention algorithm may preempt processes even if no deadlock has occurred. To prevent unnecessary preemptions, we can use a deadlock-detection algorithm. We construct a wait-for graph describing the resource-allocation state. Since we are assuming only a single resource of each type, a cycle in the wait-for graph represents a deadlock.

The main problem in a distributed system is deciding how to maintain the wait-for graph. We illustrate this problem by describing several common techniques to deal with this issue. These schemes require each site to keep a local wait-for graph. The nodes of the graph correspond to all the processes (local as well as nonlocal) currently holding or requesting any of the resources local to that site. For example, in Figure 18.3 we have a system consisting of two sites, each maintaining its local wait-for graph. Note that processes \( P_2 \) and \( P_3 \) appear in both graphs, indicating that the processes have requested resources at both sites.

These local wait-for graphs are constructed in the usual manner for local processes and resources. When a process \( P_i \) in site \( S_i \) needs a resource held by process \( P_j \) in site \( S_2 \), a request message is sent by \( P_i \) to site \( S_2 \). The edge \( P_i \rightarrow P_j \) is then inserted in the local wait-for graph of site \( S_2 \).

![Figure 18.3 Two local wait-for graphs.](image)
Figure 18.7 Augmented local wait-for graph in site $S_2$ of Figure 18.6.

$$P_{cx} \rightarrow P_{K_1} \rightarrow P_{K_2} \rightarrow \ldots \rightarrow P_{K_n} \rightarrow P_{ex},$$

it sends a deadlock-detection message to another site only if

$$ID(P_{K_n}) < ID(P_{K_1}).$$

Otherwise, site $S_k$ continues its normal execution, leaving the burden of initiating the deadlock-detection algorithm to some other site.

To illustrate this scheme, we consider again the wait-for graphs maintained at sites $S_1$ and $S_2$ of Figure 18.6. Suppose that

$$ID(P_2) < ID(P_3) < ID(P_1).$$

Let both sites discover these local cycles at about the same time. The cycle in site $S_1$ is of the form

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$  

Since $ID(P_3) > ID(P_2)$, site $S_1$ does not send a deadlock-detection message to site $S_2$.

The cycle in site $S_2$ is of the form

$$P_{ex} \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_{ex}.$$  

Since $ID(P_2) < ID(P_3)$, site $S_2$ does send a deadlock-detection message to site $S_1$, which, on receiving the message, updates its local wait-for graph. Site $S_1$ then searches for a cycle in the graph and discovers that the system is in a deadlocked state.

18.6 Election Algorithms

As we pointed out in Section 18.3, many distributed algorithms employ a coordinator process that performs functions needed by the other processes in the system. These functions include enforcing mutual exclusion, maintaining a global wait-for graph for deadlock detection, replacing a lost token, and controlling an input or output device in the system. If the coordinator process fails due to the failure of the site at which it resides, the system can continue...
execution only by restarting a new copy of the coordinator on some other site. The algorithms that determine where a new copy of the coordinator should be restarted are called **election algorithms**.

Election algorithms assume that a unique priority number is associated with each active process in the system. For ease of notation, we assume that the priority number of process \( P_i \) is \( i \). To simplify our discussion, we assume a one-to-one correspondence between processes and sites and thus refer to both as processes. The coordinator is always the process with the largest priority number. Hence, when a coordinator fails, the algorithm must elect that active process with the largest priority number. This number must be sent to each active process in the system. In addition, the algorithm must provide a mechanism for a recovered process to identify the current coordinator.

In this section, we present examples of election algorithms for two different configurations of distributed systems. The first algorithm applies to systems where every process can send a message to every other process in the system. The second algorithm applies to systems organized as a ring (logically or physically). Both algorithms require \( n^2 \) messages for an election, where \( n \) is the number of processes in the system. We assume that a process that has failed knows on recovery that it has indeed failed and thus takes appropriate actions to rejoin the set of active processes.

### 18.6.1 The Bully Algorithm

Suppose that process \( P_i \) sends a request that is not answered by the coordinator within a time interval \( T \). In this situation, it is assumed that the coordinator has failed, and \( P_i \) tries to elect itself as the new coordinator. This task is completed through the following algorithm.

Process \( P_i \) sends an election message to every process with a higher priority number. Process \( P_i \) then waits for a time interval \( T \) for an answer from any one of these processes.

If no response is received within time \( T \), \( P_i \) assumes that all processes with numbers greater than \( i \) have failed and elects itself the new coordinator. Process \( P_i \) restarts a new copy of the coordinator and sends a message to inform all active processes with priority numbers less than \( i \) that \( P_i \) is the new coordinator.

However, if an answer is received, \( P_i \) begins a time interval \( T' \), waiting to receive a message informing it that a process with a higher priority number has been elected. (That is, some other process is electing itself coordinator and should report the results within time \( T' \).) If no message is sent within \( T' \), then the process with a higher number is assumed to have failed, and process \( P_i \) should restart the algorithm.

If \( P_i \) is not the coordinator, then, at any time during execution, \( P_i \) may receive one of the following two messages from process \( P_j \):

1. \( P_j \) is the new coordinator \( (j > i) \). Process \( P_i \), in turn, records this information.
2. \( P_j \) has started an election \( (j < i) \). Process \( P_i \) sends a response to \( P_j \) and begins its own election algorithm, provided that \( P_i \) has not already initiated such an election.

The process that completes its algorithm has the highest number and is elected as the coordinator. It has sent its number to all active processes with smaller
a time interval during which it is willing to wait for an acknowledgment message from \( P_i \). When \( P_j \) receives the message, it immediately sends an acknowledgment to \( P_i \). If \( P_i \) receives the acknowledgment message within the specified time interval, it can safely conclude that \( P_j \) has received its message. If, however, a time-out occurs, then \( P_i \) needs to retransmit its message and wait for an acknowledgment. This procedure continues until \( P_i \) either gets the acknowledgment message back or is notified by the system that site \( S_i \) is down. In the first case, it will compute \( S_i \); in the latter case, it will compute \( F \). Note that, if these are the only two viable alternatives, \( P_i \) must wait until it has been notified that one of the situations has occurred.

Suppose now that \( P_j \) also needs to know that \( P_i \) has received its acknowledgment message, so that it can decide how to proceed with its computation. For example, \( P_j \) may want to compute \( \text{foo} \) only if it is assured that \( P_i \) got its acknowledgment. In other words, \( P_i \) and \( P_j \) will compute \( \text{foo} \) if and only if both have agreed on it. It turns out that, in the presence of failure, it is not possible to accomplish this task. More precisely, it is not possible in a distributed environment for processes \( P_i \) and \( P_j \) to agree completely on their respective states.

To prove this claim, let us suppose that a minimal sequence of message transfers exists such that, after the messages have been delivered, both processes agree to compute \( \text{foo} \). Let \( m \) be the last message sent by \( P_i \) to \( P_j \). Since \( P_i \) does not know whether its message will arrive at \( P_j \) (since the message may be lost due to a failure), \( P_i \) will execute \( \text{foo} \) regardless of the outcome of the message delivery. Thus, \( m \) could be removed from the sequence without affecting the decision procedure. Hence, the original sequence was not minimal, contradicting our assumption and showing that there is no sequence. The processes can never be sure that both will compute \( \text{foo} \).

### 18.7.2 Faulty Processes

Now let us assume that the communication medium is reliable but that processes can fail in unpredictable ways. Consider a system of \( n \) processes, of which no more than \( m \) are faulty. Suppose that each process \( P_i \) has some private value of \( V_i \). We wish to devise an algorithm that allows each nonfaulty process \( P_i \) to construct a vector \( X_i = (A_{i,1}, A_{i,2}, \ldots, A_{i,n}) \) such that the following conditions exist:

1. If \( P_j \) is a nonfaulty process, then \( A_{i, j} = V_j \).
2. If \( P_i \) and \( P_j \) are both nonfaulty processes, then \( X_i = X_j \).

There are many solutions to this problem, and they share the following properties:

1. A correct algorithm can be devised only if \( n > 3 \times m + 1 \).
2. The worst-case delay for reaching agreement is proportionate to \( m + 1 \) message-passing delays.
3. The number of messages required for reaching agreement is large. No single process is trustworthy, so all processes must collect all information and make their own decisions.
Real-Time Systems

Our coverage of operating-system issues thus far has focused mainly on general-purpose computing systems (for example, desktop and server systems). In this chapter, we turn our attention to real-time computing systems. The requirements of real-time systems differ from those of any of the systems we have described, largely because real-time systems must produce results within certain deadlines. In this chapter, we provide an overview of real-time computer systems and describe how real-time operating systems must be constructed to meet the stringent timing requirements of these systems.

CHAPTER OBJECTIVES

- To explain the timing requirements of real-time systems.
- To distinguish between hard and soft real-time systems.
- To discuss the defining characteristics of real-time systems,
- To describe scheduling algorithms for hard real-time systems.

19.1 Overview

A real-time system is a computer system that requires not only that the computing results be "correct" but also that the results be produced within a specified deadline period. Results produced after the deadline has passed—even if correct—may be of no real value. To illustrate, consider an autonomous robot that delivers mail in an office complex. If its vision-control system identifies a wall after the robot has walked into it, despite correctly identifying the wall, the system has not met its requirement. Contrast this timing requirement with the much less strict demands of other systems. In an interactive desktop computer system, it is desirable to provide a quick response time to the interactive user, but it is not mandatory to do so. Some systems—such as a batch-processing system—may have no timing requirements whatsoever.

Real-time systems executing on traditional computer hardware are used in a wide range of applications. In addition, many real-time systems are
19.5.3 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 there is a total of \( T = 100 \) shares 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 only admit a client requesting a particular number of shares if there are sufficient shares available. In our current example, we have allocated \( 50 + 15 + 20 = 75 \) 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.

19.5.4 Pthread Scheduling

The POSIX standard also provides extensions for real-time computing—POSIX.1b. In this section, we cover some of the POSIX Pthread API related to scheduling real-time threads. Pthread 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 (for round-robin) is similar to SCHED_FIFO except that it provides time slicing among threads of equal priority. Pthreads provides an additional scheduling class—SCHED_OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The Pthread API specifies the following two functions for getting and setting the scheduling policy:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_setsched_policy(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 a pointer to an integer that is set to the current scheduling policy (for `pthread_attr_getsched_policy()`) or an integer value—SCHED_FIFO, SCHED_RR, or SCHED_OTHER—for the `pthread_attr_setsched_policy()` function. Both functions return non-zero values if an error occurs.
include only required features, thereby minimizing the size—or footprint—of the operating system.

VxWorks takes an interesting approach to memory management, supporting two levels of virtual memory. The first level, which is quite simple, allows control of the cache on a per-page basis. This policy enables an application to specify certain pages as non-cacheable. When data are being shared by separate tasks running on a multiprocessor architecture, it is possible that shared data can reside in separate caches local to individual processors. Unless an architecture supports a cache-coherency policy to ensure that the same data residing in two caches will not be different, such shared data should not be cached and should instead reside only in main memory so that all tasks maintain a consistent view of the data.

The second level of virtual memory requires the optional virtual memory component VxVMI (Figure 19.12), along with processor support for a memory management unit (MMU). By loading this optional component on systems with an MMU, VxWorks allows a task to mark certain data areas as private. A data area marked as private may only be accessed by the task it belongs to. Furthermore, VxWorks allows pages containing kernel code along with the interrupt vector to be declared as read-only. This is useful, as VxWorks does not distinguish between user and kernel modes; all applications run in kernel mode, giving an application access to the entire address space of the system.

19.7 Summary

A real-time system is a computer system requiring that results arrive within a deadline period; results arriving after the deadline has passed are useless. Many real-time systems are embedded in consumer and industrial devices. There are two types of real-time systems: soft and hard real-time systems. Soft real-time systems are the least restrictive, assigning real-time tasks higher scheduling priority than other tasks. Hard real-time systems must guarantee that real-time tasks are serviced within their deadline periods. In addition to strict timing requirements, real-time systems can further be characterized as having only a single purpose and running on small, inexpensive devices.

To meet timing requirements, real-time operating systems must employ various techniques. The scheduler for a real-time operating system must support a priority-based algorithm with preemption. Furthermore, the operating system must allow tasks running in the kernel to be preempted in favor of higher-priority real-time tasks. Real-time operating systems also address specific timing issues by minimizing both interrupt and dispatch latency.

Real-time scheduling algorithms include rate-monotonic and earliest-deadline-first scheduling. Rate-monotonic scheduling assigns tasks that require the CPU more often a higher priority than tasks that require the CPU less often. Earliest-deadline-first scheduling assigns priority according to upcoming deadlines—the earlier the deadline, the higher the priority. Proportional share scheduling uses a technique of dividing up processor time into shares and assigning each process a number of shares, thus guaranteeing each process its proportional share of CPU time. The Pthread API provides various features for scheduling real-time threads as well.
frame requires 800 x 600 x 24 = 11,520,000 bits of data. If the frames are displayed at a rate of 30 frames per second, a bandwidth in excess of 345 Mbps is required.

3. Multimedia applications are sensitive to timing delays during playback. Once a continuous-media file is delivered to a client, delivery must continue at a certain rate during playback of the media; otherwise, the listener or viewer will be subjected to pauses during the presentation.

20.1.3 Operating-System Issues

For a computer system to deliver continuous-media data, it must guarantee the specific rate and timing requirements—also known as quality of service, or QoS, requirements—of continuous media.

Providing these QoS guarantees affects several components in a computer system and influences such operating-system issues as CPU scheduling, disk scheduling, and network management. Specific examples include the following:

1. Compression and decoding may require significant CPU processing.

2. Multimedia tasks must be scheduled with certain priorities to ensure meeting the deadline requirement of continuous media.

3. Similarly, file systems must be efficient to meet the rate requirements of continuous media.

4. Network protocols must support bandwidth requirements while minimizing delay and jitter.

In later sections, we explore these and several other issues related to QoS. First, however, we provide an overview of various techniques for compressing multimedia data. As suggested above, compression makes significant demands on the CPU.

20.2 Compression

Because of the size and rate requirements of multimedia systems, multimedia files are often compressed from their original form to a much smaller form. Once a file has been compressed, it takes up less space for storage and can be delivered to a client more quickly. Compression is particularly important when the content is being streamed across a network connection. In discussing file compression, we often refer to the compression ratio, which is the ratio of the original file size to the size of the compressed file. For example, an 800-KB file that is compressed to 100 KB has a compression ratio of 8:1.

Once a file has been compressed (encoded), it must be decompressed (decoded) before it can be accessed. A feature of the algorithm used to compress the file affects the later decompression. Compression algorithms are classified as either lossy or lossless. With lossy compression, some of the original data are lost when the file is decoded, whereas lossless compression ensures that the compressed file can always be restored back to its original form. In general, lossy techniques provide much higher compression ratios. Obviously, though,
The timing requirements of multimedia data are known as quality-of-service requirements, and conventional operating systems often cannot make quality-of-service guarantees. To provide quality of service, multimedia systems must provide a form of admission control whereby a system accepts a request only if it can meet the quality-of-service level specified by the request. Providing quality-of-service guarantees requires evaluating how an operating system performs CPU scheduling, disk scheduling, and network management. Both CPU and disk scheduling typically use the deadline requirements of a continuous-media task as a scheduling criterion. Network management requires the use of protocols that handle delay and jitter caused by the network as well as allowing a client to pause or move to different positions in the stream during playback.

Exercises

20.1 Provide examples of multimedia applications that are delivered over the Internet.

20.2 Distinguish between progressive download and real-time streaming.

20.3 Which of the following types of real-time streaming applications can tolerate delay? Which can tolerate jitter?
   - Live real-time streaming
   - On-demand real-time streaming

20.4 Discuss what techniques could be used to meet quality-of-service requirements for multimedia applications in the following components of a system:
   - Process scheduler
   - Disk scheduler
   - Memory manager

20.5 Explain why the traditional Internet protocols for transmitting data are not sufficient to provide the quality-of-service guarantees required for a multimedia system. Discuss what changes are required to provide the QoS guarantees.

20.6 Assume that a digital video file is being displayed at a rate of 30 frames per second; the resolution of each frame is 640 x 480, and 24 bits are being used to represent each color. Assuming that no compression is being used, what is the bandwidth necessary to deliver this file? Next, assuming that the file has been compressed at a ratio of 200 : 1, what is the bandwidth necessary to deliver the compressed file?

20.7 A multimedia application consists of a set containing 100 images, 10 minutes of video, and 10 minutes of audio. The compressed sizes of the images, video, and audio are 500 MB, 550 MB, and 8 MB, respectively. The images were compressed at a ratio of 15 : 1, and the video and
21.3 Kernel Modules

The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand. These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run. In theory, there is no restriction on what a kernel module is allowed to do; typically, a module might implement a device driver, a file system, or a networking protocol.

Kernel modules are convenient for several reasons. Linux's source code is free, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot to load that new functionality; however, recompiling, relinking, and reloading the entire kernel is a cumbersome cycle to undertake when you are developing a new driver. If you use kernel modules, you do not have to make a new kernel to test a new driver—the driver can be compiled on its own and loaded into the already-running kernel. Of course, once a new driver is written, it can be distributed as a module so that other users can benefit from it without having to rebuild their kernels.

This latter point has another implication. Because it is covered by the GPL license, the Linux kernel cannot be released with proprietary components added to it, unless those new components are also released under the GPL and the source code for them is made available on demand. The kernel's module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that would not be distributable under the GPL.

Kernel modules allow a Linux system to be set up with a standard, minimal kernel without any extra device drivers built in. Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use. For example, a CD-ROM driver might be loaded when a CD is mounted and unloaded from memory when the CD is dismounted from the file system.

The module support under Linux has three components:

1. The **module management** allows modules to be loaded into memory and to talk to the rest of the kernel.

2. The **driver registration** allows modules to tell the rest of the kernel that a new driver has become available.

3. A **conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

21.3.1 Module Management

Loading a module requires more than just loading its binary contents into kernel memory. The system must also make sure that any references the module makes to kernel symbols or entry points are updated to point to the correct locations in the kernel's address space. Linux deals with this reference updating by splitting the job of module loading into two separate sections: the management of sections of module code in kernel memory and the handling of symbols that modules are allowed to reference.
to decide how to proceed. It may fail its initialization and request that it be unloaded if it cannot continue, or it may carry on, using alternative hardware resources.

21.4 Process Management

A process is the basic context within which all user-requested activity is serviced within the operating system. To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX. Linux operates differently from UNIX in a few key places, however. In this section, we review the traditional UNIX process model from Section A.3.2 and introduce Linux's own threading model.

21.4.1 The fork() and exec() Process Model

The basic principle of UNIX process management is to separate two operations: the creation of a process and the running of a new program. A new process is created by the fork() system call, and a new program is started by a call to exec(). These are two distinctly separate functions. A new process may be created with fork() without a new program being run—the new subprocess simply continues to execute exactly the same program that the first, parent process was running. Equally, running a new program does not require that a new process be created first. Any process may call exec() at any time. The currently running program is immediately terminated, and the new program starts executing in the context of the existing process.

This model has the advantage of great simplicity. Rather than having to specify every detail of the environment of a new program in the system call that runs that program, new programs simply run in their existing environment. If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original program in a child process, make any system calls it requires to modify that child process before finally executing the new program.

Under UNIX, then, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program. Under Linux, we can break down this context into a number of specific sections. Broadly, process properties fall into three groups: the process identity, environment, and context.

21.4.1.1 Process Identity

A process identity consists mainly of the following items:

- **Process ID (PID).** Each process has a unique identifier. PIDs are used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process. Additional identifiers associate the process with a process group (typically, a tree of processes forked by a single user command) and login session.

- ** Credentials.** Each process must have an associated user ID and one or more group IDs (user groups are discussed in Section 10.6.2) that determine the rights of a process to access system resources and files.
• **Personality.** Process personalities are not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can modify slightly the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain flavors of UNIX.

Most of these identifiers are under limited control of the process itself. The process group and session identifiers can be changed if the process wants to start a new group or session. Its credentials can be changed, subject to appropriate security checks. However, the primary PID of a process is unchangeable and uniquely identifies that process until termination.

### 21.4.1.2 Process Environment

A process's environment is inherited from its parent and is composed of two null-terminated vectors: the argument vector and the environment vector. The **argument vector** simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself. The **environment vector** is a list of "NAME=VALUE" pairs that associate named environment variables with arbitrary textual values. The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.

The argument and environment vectors are not altered when a new process is created: The new child process will inherit the environment that its parent possesses. However, a completely new environment is set up when a new program is invoked. On calling **exec**(), a process must supply the environment for the new program. The kernel passes these environment variables to the next program, replacing the process's current environment. The kernel otherwise leaves the environment and command-line vectors alone—their interpretation is left entirely to the user-mode libraries and applications.

The passing of environment variables from one process to the next and the inheriting of these variables by the children of a process provide flexible ways to pass information to components of the user-mode system software. Various important environment variables have conventional meanings to related parts of the system software. For example, the **TERM** variable is set up to name the type of terminal connected to a user's login session; many programs use this variable to determine how to perform operations on the user's display, such as moving the cursor and scrolling a region of text. Programs with multilingual support use the **LANG** variable to determine in which language to display system messages for programs that include multilingual support.

The environment-variable mechanism custom tailors the operating system on a per-process basis, rather than for the system as a whole. Users can choose their own languages or select their own editors independently of one another.

### 21.4.1.3 Process Context

The process identity and environment properties are usually set up when a process is created and not changed until that process exits. A process may choose to change some aspects of its identity if it needs to do so, or it may alter its environment. In contrast, process context is the state of the running program at any one time; it changes constantly. Process context includes the following parts.
processor affinity and load balancing, as well as maintaining fairness and support for interactive tasks.

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a real-time range from 0 to 99 and a nice value ranging from 100 to 140. These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.

Unlike schedulers for many other systems, Linux assigns higher-priority tasks longer time quanta and vice-versa. Because of the unique nature of the scheduler, this is appropriate for Linux, as we shall soon see. The relationship between priorities and time-slice length is shown in Figure 21.2.

A runnable task is considered eligible for execution on the CPU so long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quanta. The kernel maintains a list of all runnable tasks in a runqueue data structure. Because of its support for SMP, each processor maintains its own runqueue and schedules itself independently. Each runqueue contains two priority arrays—active and expired. The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays includes a list of tasks indexed according to priority (Figure 21.3). The scheduler chooses the task with the highest priority from the active array for execution on the CPU. On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own runqueue structure. When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged as the expired array becomes the active array and vice-versa.

Tasks are assigned dynamic priorities that are based on the nice value plus or minus up to the value 5 based upon the interactivity of the task. Whether a value is added to or subtracted from a task's nice value depends on the interactivity of the task. A task's interactivity is determined by how long it has been sleeping while waiting for I/O. Tasks that are more interactive typically have longer sleep times and therefore are more likely to have an adjustment closer to -5, as the scheduler favors such interactive tasks. Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have their priorities lowered.
The recalculation of a task’s dynamic priority occurs when the task has exhausted its time quantum and is to be moved to the expired array. Thus, when the two arrays are exchanged, all tasks in the new active array have been assigned new priorities and corresponding time slices.

Linux’s real-time scheduling is simpler still. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first-served (FCFS) and round-robin (Sections 5.3.1 and 5.3.4, respectively). In both cases, each process has a priority in addition to its scheduling class. Processes of different priorities can compete with one another to some extent in time-sharing scheduling; in real-time scheduling, however, the scheduler always runs the process with the highest priority. Thus, processes of equal priority, it runs the process that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves. Unlike routine time-sharing tasks, real-time tasks are assigned static priorities.

Linux’s real-time scheduling is soft—rather than hard—real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable.

21.5.2 Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules processes. A request for kernel-mode execution can occur in two ways. A running program may request an operating-system service, either explicitly via a system call or implicitly—for example, when a page fault occurs. Alternatively, a device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem posed to the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption. This fact relates to the idea of critical sections—portions of code that access shared data and that must not be allowed to execute concurrently. As a result, kernel synchronization involves much more than just process
improved performance. The allocator records the fact that a page has been paged out to disk by using a feature of the page tables on modern processors: The page-table entry’s page-not-present bit is set, allowing the rest of the page-table entry to be filled with an index identifying where the page has been written.

21.6.2.4 Kernel Virtual Memory

Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process. The page-table entries that map to these kernel pages are marked as protected, so that the pages are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions. The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run. The core of the kernel, along with all pages allocated by the normal page allocator, resides in this region.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose. Page-table entries in the address range can be modified by the kernel to point to unallocated areas of memory. The kernel provides a pair of facilities for user processes to use this virtual memory. The vmalloc() function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The vremap() function maps a sequence of linear addresses to point to an area of memory used by a device driver for memory-mapped I/O.

21.6.3 Execution and Loading of User Programs

The Linux kernel’s execution of user programs is triggered by a call to the exec() system call. This call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program. The first job of this system service is to verify that the calling process has permission rights to the file being executed. Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.

There is no single routine in Linux for loading a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an exec() system call is made. The initial reason for this loader table was that, between the releases of the 1.0 and 1.2 kernels, the standard format for Linux's binary files was changed. Older Linux kernels understood the a.out format for binary files—a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern ELF format, now supported by most current UNIX implementations. ELF has a number of advantages over a.out, including flexibility and extensibility: New sections can be added to an ELF binary (for example, to add extra debugging information) without causing...
as easily as it can wait for a signal, but semaphores have two advantages: Large numbers of semaphores can be shared among multiple independent processes, and operations on multiple semaphores can be performed atomically. Internally, the standard Linux wait queue mechanism synchronizes processes that are communicating with semaphores.

21.9.2 Passing of Data Among Processes

Linux offers several mechanisms for passing data among processes. The standard UNIX pipe mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other. Under Linux, pipes appear as just another type of inode to virtual-file-system software, and each pipe has a pair of wait queues to synchronize the reader and writer. UNIX also defines a set of networking facilities that can send streams of data to both local and remote processes. Networking is covered in Section 21.10.

Two other methods of sharing data among processes are available. First, shared memory offers an extremely fast way to communicate large or small amounts of data; any data written by one process to a shared-memory region can be read immediately by any other process that has mapped that region into its address space. The main disadvantage of shared memory is that, on its own, it offers no synchronization. A process can neither ask the operating system whether a piece of shared memory has been written to nor suspend execution until such a write occurs. Shared memory becomes particularly powerful when used in conjunction with another interprocess-communication mechanism that provides the missing synchronization.

A shared-memory region in Linux is a persistent object that can be created or deleted by processes. Such an object is treated as though it were a small independent address space. The Linux paging algorithms can elect to page out to disk shared-memory pages, just as they can page out a process's data pages. The shared-memory object acts as a backing store for shared-memory regions, just as a file can act as a backing store for a memory-mapped memory region. When a file is mapped into a virtual-address-space region, then any page faults that occur cause the appropriate page of the file to be mapped into virtual memory. Similarly, shared-memory mappings direct page faults to map in pages from a persistent shared-memory object. Also just as for files, shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

21.10 Network Structure

Networking is a key area of functionality for Linux. Not only does Linux support the standard Internet protocols used for most UNIX-to-UNIX communications, but it also implements a number of protocols native to other, non-UNIX operating systems. In particular, since Linux was originally implemented primarily on PCs, rather than on large workstations or on server-class systems, it supports many of the protocols typically used on PC networks, such as AppleTalk and IPX.
Packets (skbufs) arriving at the networking stack's protocol software are expected to be already tagged with an internal identifier indicating to which protocol the packet is relevant. Different networking-device drivers encode the protocol type in different ways over their communications media; thus, the protocol for incoming data must be identified in the device driver. The device driver uses a hash table of known networking-protocol identifiers to look up the appropriate protocol and passes the packet to that protocol. New protocols can be added to the hash table as kernel-loadable modules.

Incoming IP packets are delivered to the IP driver. The job of this layer is to perform routing. After deciding where the packet is destined, it forwards the packet to the appropriate internal protocol driver to be delivered locally or injects it back into a selected network-device-driver queue to be forwarded to another host. It performs the routing decision using two tables: the persistent forwarding information base (FIB) and a cache of recent routing decisions. The FIB holds routing-configuration information and can specify routes based either on a specific destination address or on a wildcard representing multiple destinations. The FIB is organized as a set of hash tables indexed by destination address; the tables representing the most specific routes are always searched first. Successful lookups from this table are added to the route-caching table, which caches routes only by specific destination. Wildcards are stored in the cache, so lookups can be made quickly. An entry in the route cache expires after a fixed period with no hits.

At various stages, the IP software passes packets to a separate section of code for firewall management—selective filtering of packets according to arbitrary criteria, usually for security purposes. The firewall manager maintains a number of separate firewall chains and allows an skbuff to be matched against any chain. Chains are reserved for separate purposes: One is used for forwarded packets, one for packets being input to this host, and one for data generated at this host. Each chain is held as an ordered list of rules, where a rule specifies one of a number of possible firewall-decision functions plus some arbitrary data to match against.

Two other functions performed by the IP driver are disassembly and reassembly of large packets. If an outgoing packet is too large to be queued to a device, it is simply split up into smaller fragments, which are all queued to the driver. At the receiving host, these fragments must be reassembled. The IP driver maintains an ipfrag object for each fragment awaiting reassembly and an ipq for each datagram being assembled. Incoming fragments are matched against each known ipq. If a match is found, the fragment is added to it; otherwise, a new ipq is created. Once the final fragment has arrived for a ipq, a completely new skbuff is constructed to hold the new packet, and this packet is passed back into the IP driver.

Packets identified by the IP as destined for this host are passed on to one of the other protocol drivers. The UDP and TCP protocols share a means of associating packets with source and destination sockets: Each connected pair of sockets is uniquely identified by its source and destination addresses and by the source and destination port numbers. The socket lists are linked onto hash tables keyed on these four address-port values for socket lookup on incoming packets. The TCP protocol has to deal with unreliable connections, so it maintains ordered lists of unacknowledged outgoing packets to retransmit.
Windows API (or Win32 API), reflecting the popularity of Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at version 3.1.) Windows NT version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance, with the side effect of decreased system reliability. Although previous versions of NT had been ported to other microprocessor architectures, the Windows 2000 version, released in February 2000, discontinued support for other than Intel (and compatible) processors due to marketplace factors. Windows 2000 incorporated significant changes over Windows NT. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In 2002, the server versions of Windows XP became available (called Windows .Net Server). Windows XP updates the graphical user interface (GUI) with a visual design that takes advantage of more recent hardware advances and many new ease-of-use features. New features have been added to automatically repair problems with applications and the operating system itself. Windows XP provides better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photographs/video) dramatic performance improvements both for the desktop and large multiprocessors, and better reliability and security than ever before.

Windows .Net Server has a client–server architecture (like Mach) to implement multiple operating-system personalities, such as Win32 API and POSIX, with user-level processes called subsystems. The subsystem architecture allows enhancements to be made to one operating-system personality without affecting the application compatibility of any others.

Windows XP is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the graphical user interface via the Windows terminal server. The server versions of Windows XP support simultaneous terminal server sessions from Windows desktop systems. The desktop versions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called fast user switching, allows users to preempt each other at the console of a PC without having to log off and onto the system.

Windows XP is the first version of Windows to ship a 64-bit version. The native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate—so the major extension to 64-bit in Windows XP is support for large addresses.

There are two desktop versions of Windows XP. Windows XP Professional is the premium desktop system for power users at work and at home. For home users migrating from Windows 95/98, Windows XP Personal provides the reliability and ease of use of Windows XP, but lacks the more advanced features needed to work seamlessly with Active Directory or run POSIX applications.

The members of the Windows .Net Server family use the same core components as the desktop versions but add a range of features needed for
uses such as webserver farms, print/file servers, clustered systems, and large datacenter machines. The large datacenter machines can have up to 64 GB of memory and 32 processors on IA32 systems and 128 GB and 64 processors on IA64 systems.

22.2 Design Principles

Microsoft's design goals for Windows XP include security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support.

22.2.1 Security

Windows XP security goals required more than just adherence to the design standards that enabled Windows NT 4.0 to receive a C-2 security classification from the U.S. government (which signifies a moderate level of protection from defective software and malicious attacks). Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities.

22.2.2 Reliability

Windows 2000 was the most reliable, stable operating system Microsoft had ever shipped to that point. Much of this reliability came from maturity in the source code, extensive testing of the system, and automatic detection of many serious errors in drivers. The reliability requirements for Windows XP were even more stringent. Microsoft used extensive manual and automatic code review to identify over 63,000 lines in the source files that might contain issues not detected by testing and then set about reviewing each area to verify that the code was indeed correct.

Windows XP extends driver verification to catch more subtle bugs, improves the facilities for catching programming errors in user-level code, and subjects third-party applications, drivers, and devices to a rigorous certification process. Furthermore, Windows XP adds new facilities for monitoring the health of the PC, including downloading fixes for problems before they are encountered by users. The perceived reliability of Windows XP was also improved by making the graphical user interface easier to use through better visual design, simpler menus, and measured improvements in the ease with which users can discover how to perform common tasks.

22.2.3 Windows and POSIX Application Compatibility

Windows XP is not only an update of Windows 2000; it is a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP include a much higher compatibility with consumer applications that run on Windows 95/98. Application compatibility is difficult to achieve because each application checks for a particular version of Windows, may have some dependence on the
priority. Lowering the thread's 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 the device for which the thread was waiting; for example, a thread waiting for keyboard I/O would get a large priority 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 using a 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 time-sharing operating systems, including UNIX. In addition, the thread associated with the user’s active GUI window receives a priority boost to enhance its response time.

Scheduling occurs when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access. Windows XP is not a hard real-time operating system, however, because it does not guarantee that a real-time thread will start to execute within a particular time limit.

22.3.2.3 Implementation of Synchronization Primitives

Key operating system data structures are managed as objects using common facilities for allocation, reference counting, and security. Dispatcher objects control dispatching and synchronization in the system. Examples of these objects are events, mutants, mutexes, semaphores, processes, threads, and timers. The event object is used to record an event occurrence and to synchronize the latter with some action. Notification events signal all waiting threads, and synchronization events signal a single waiting thread. The mutant provides kernel-mode or user-mode mutual exclusion with the notion of ownership. The mutex, available only in kernel mode, provides deadlock-free mutual exclusion. A semaphore object acts as a counter or gate to control the number of threads that access a resource. The thread object is the entity that is scheduled by the kernel dispatcher and is associated with a process object, which encapsulates a virtual address space. Timer objects are used to keep track of time and to signal timeouts when operations take too long and need to be interrupted or when a periodic activity needs to be scheduled.

Many of the dispatcher objects are accessed from user mode via an open operation that returns a handle. The user-mode code polls and/or waits on handles to synchronize with other threads as well as the operating system (see Section 22.7.1).

22.3.2.4 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: asynchronous procedure calls and deferred procedure calls. Asynchronous procedure calls (APCs) break into an executing thread and call a procedure. APCs are used to begin execution of a new thread, terminate processes, and deliver notification that an asynchronous (I/O) has completed. APCs are queued to specific threads
A name can be either permanent or temporary. A permanent name represents an entity, such as a disk drive, that remains even if no process is accessing it. A temporary name exists only while a process holds a handle to the object.

Object names are structured like file path names in MS-DOS and UNIX. Name space directories are represented by a directory object that contains the names of all the objects in the directory. The object name space is extended by the addition of device objects representing volumes containing file systems.

Objects are manipulated by a set of virtual functions with implementations provided for each object type: create(), open(), close(), delete(), query_name(), parse(), and security(). The latter three objects need explanation:

- query_name() is called when a thread has a reference to an object but wants to know the object's name.
- parse() is used by the object manager to search for an object given the object's name.
- security() is called to make security checks on all object operations, such as when a process opens or closes an object, makes changes to the security descriptor, or duplicates a handle for an object.

The parse procedure is used to extend the abstract name space to include files. The translation of a path name to a file object begins at the root of the abstract name space. Path-name components are separated by backslash characters (\') rather than the slashes (/') used in UNIX. Each component is looked up in the current parse directory of the name space. Internal nodes within the name space are either directories or symbolic links. If a leaf object is found and there are no path-name components remaining, the leaf object is returned. Otherwise, the leaf object's parse procedure is invoked with the remaining path name.

Parse procedures are only used with a small number of objects belonging to the Windows GUI, the configuration manager (registry), and—most notably—device objects representing file systems.

The parse procedure for the device object type allocates a file object and initiates an open or create I/O operation on the file system. If successful, the file object fields are filled in to describe the file.

In summary, the path name to a file is used to traverse the object-manager namespace, translating the original absolute path name into a (device object, relative path name) pair. This pair is then passed to the file system via the I/O manager, which fills in the file object. The file object itself has no name but is referred to by a handle.

UNIX file systems have symbolic links that permit multiple nicknames — or aliases — for the same file. The symbolic-link object implemented by the Windows XP object manager is used within the abstract name space, not to provide file aliases on a file system. Even so, symbolic links are very useful. They are used to organize the name space, similar to the organization of the /devices directory in UNIX. They are also used to map standard MS-DOS drive letters to drive names. Drive letters are symbolic links that can be remapped to suit the convenience of the user or administrator.
Drive letters are one place where the abstract name space in Windows XP is not global. Each logged-on user has his or her own set of drive letters so that users can avoid interfering with one another. In contrast, terminal server sessions share all processes within a session. BaseNamedObjects contain the named objects created by most applications.

Although the name space is not directly visible across a network, the object manager's parse() method is used to help access a named object on another system. When a process attempts to open an object that resides on a remote computer, the object manager calls the parse method for the device object corresponding to a network redirector. This results in an I/O operation that accesses the file across the network.

Objects are instances of an object type. The object type specifies how instances are to be allocated, the definitions of the data fields, and the implementation of the standard set of virtual functions used for all objects. These functions implement operations such as mapping names to objects, closing and deleting, and applying security.

The object manager keeps track of two counts for each object. The pointer count is the number of distinct references made to an object. User-mode code that refers to objects must keep a reference on the object to ensure that the object is not deleted while in use. The handle count is the number of handle table entries referring to an object. Each handle is also reflected in the reference count.

When a handle to an object is closed, the object's close routine is called. In the case of file objects, this call causes the I/O manager to do a cleanup operation at the close of the last handle. The cleanup operation tells the file system that the file is no longer accessible by user mode so that sharing restrictions, range locks, and other information specific to the corresponding open routine can be removed.

Each handle close removes a reference from the pointer count, but internal system components may retain additional references. When the final reference is removed, the object's delete procedure is called. Again using file objects as an example, the delete procedure causes the I/O manager to send the file system a close operation on the file object. This causes the file system to deallocate any internal data structures that were allocated for the file object.

After the delete procedure for a temporary object completes, the object is deleted from memory. Objects can be made permanent (at least with respect to the current boot of the system) by asking the object manager to take an extra reference against the object. Thus, permanent objects are not deleted even when the last reference outside the object manager is removed. When a permanent object is made temporary again, the object manager removes the extra reference. If this was the last reference, the object is deleted. Permanent objects are rare, used mostly for devices, drive-letter mappings, and the directory and symbolic link objects.

The job of the object manager is to supervise the use of all managed objects. When a thread wants to use an object, it calls the object manager's open() method to get a reference to the object. If the object is being opened from a user-mode API, the reference is inserted into the process's object table, and a handle is returned.

A process gets a handle by creating an object, by opening an existing object, by receiving a duplicated handle from another process, or by inheriting a handle from a parent process, similar to the way a UNIX process gets a file...
partition. A volume may occupy a portion of a disk, may occupy an entire
disk, or may span several disks.

NTFS does not deal with individual sectors of a disk but instead uses clusters
as the units of disk allocation. A cluster is a number of disk sectors that is a
power of 2. The cluster size is configured when an NTFS file system is formatted.
The default cluster size is the sector size for volumes up to 512 MB, 1 KB for
volumes up to 1 GB, 2 KB for volumes up to 2 GB, and 4 KB for larger volumes.
This cluster size is much smaller than that for the 16-bit FAT file system, and
the small size reduces the amount of internal fragmentation. As an example,
consider a 1.6-GB disk with 16,000 files. If you use a FAT-16 file system, 400 MB
may be lost to internal fragmentation because the cluster size is 32 KB. Under
NTFS, only 17 MB would be lost when storing the same files.

NTFS uses logical cluster numbers (LCNs) as disk addresses. It assigns them
by numbering clusters from the beginning of the disk to the end. Using this
scheme, the system can calculate a physical disk offset (in bytes) by multiplying
the LCN by the cluster size.

A file in NTFS is not a simple byte stream as it is in MS-DOS or UNIX; rather, it
is a structured object consisting of typed attributes. Each attribute of a file is an
independent byte stream that can be created, deleted, read, and written. Some
attribute types are standard for all files, including the file name (or names,
if the file has aliases, such as an MS-DOS shortname), the creation time,
and the security descriptor that specifies access control. User data is stored in data
attributes.

Most traditional data files have an unnamed data attribute that contains all
the file's data. However, additional data streams can be created with explicit
names. For instance, in Macintosh files stored on a Windows XP server, the
resource fork is a named data stream. The IProp interfaces of the Component
Object Model (COM) use a named data stream to store properties on ordinary
files, including thumbnails of images. In general, attributes may be added as
necessary and are accessed using file-name:attribute syntax. NTFS returns the
size of the unnamed attribute only in response to file-query operations, such
as when running the dir command.

Every file in NTFS is described by one or more records in an array stored in a
special file called the master file table (MFT). The size of a record is determined
when the file system is created; it ranges from 1 to 4 KB. Small attributes
are stored in the MFT record itself and are called resident attributes. Large
attributes, such as the unnamed bulk data, are called nonresident attributes
and are stored in one or more contiguous extents on the disk; a pointer to
each extent is stored in the MFT record. For a small file, even the data attribute
may fit inside the MFT record. If a file has many attributes—or if it is highly
fragmented, so that many pointers are needed to point to all the fragments
—one record in the MFT might not be large enough. In this case, the file is
described by a record called the base file record, which contains pointers to
overflow records that hold the additional pointers and attributes.

Each file in an NTFS volume has a unique ID called a file reference. The file
reference is a 64-bit quantity that consists of a 48-bit file number and a 16-bit
sequence number. The file number is the record number (that is, the array slot)
in the MFT that describes the file. The sequence number is incremented every
time an MFT entry is reused. The sequence number enables NTFS to perform
avoid creating this bottleneck, we spread the parity stripe over all the disks by assigning them in a round-robin style. To build a stripe set with parity, we need a minimum of three equal-sized partitions located on three separate disks.

22.5.4.4 Disk Mirroring

An even more robust scheme is called disk mirroring or RAID level 1; it is depicted in Figure 22.10. A mirror set comprises two equal-sized partitions on two disks. When an application writes data to a mirror set, FtDisk writes the data to both partitions, so that the data contents of the two partitions are identical. If one partition fails, FtDisk has another copy safely stored on the mirror. Mirror sets can also improve performance, because read requests can be split between the two mirrors, giving each mirror half of the workload. To protect against the failure of a disk controller, we can attach the two disks of a mirror set to two separate disk controllers. This arrangement is called a duplex set.

22.5.4.5 Sector Sparing and Cluster Remapping

To deal with disk sectors that go bad, FtDisk uses a hardware technique called sector sparing, and NTFS uses a software technique called cluster remapping. Sector sparing is a hardware capability provided by many disk drives. When a disk drive is formatted, it creates a map from logical block numbers to good sectors on the disk. It also leaves extra sectors unmapped, as spares. If a sector fails, FtDisk instructs the disk drive to substitute a spare. Cluster remapping is a software technique performed by the file system. If a disk block goes bad, NTFS substitutes a different, unallocated block by changing any affected pointers in the MFT. NTFS also makes a note that the bad block should never be allocated to any file.
information for the client. The client chooses one of the configurations and sends a request message to the selected DHCP server. The DHCP server responds with the IP address and configuration information it gave previously and with a lease for that address. The lease gives the client the right to use the IP address for a specified period of time. When the lease time is half expired, the client attempts to renew the lease for the address. If the lease is not renewed, the client must obtain a new one.

22.7 Programmer Interface

The Win32 API is the fundamental interface to the capabilities of Windows XP. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

22.7.1 Access to Kernel Objects

The Windows XP kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named XXX by calling the CreateXXX function to obtain a handle to XXX. This handle is unique to the process. Depending on which object is being opened, if the Create() function fails, it may return 0, or it may return a special constant named INVALID_HANDLE_VALUE. A process can close any handle by calling the CloseHandle() function, and the system may delete the object if the count of processes using the object drops to 0.

22.7.2 Sharing Objects Between Processes

Windows XP provides three ways to share objects between processes. The first way is for a child process to inherit a handle to the object. When the parent calls the CreateXXX function, the parent supplies a SECURITY_ATTRIBUTES structure with the bInheritHandle field set to TRUE. This field creates an inheritable handle. Next, the child process is created, passing a value of TRUE to the CreateProcessO function's bInheritHandle argument. Figure 22.11 shows a code sample that creates a semaphore handle inherited by a child process.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure 22.11, the child process gets the value of the handle from the first command-line argument and then shares the semaphore with the parent process.

The second way to share objects is for one process to give the object a name when the object is created and for the second process to open the name. This method has two drawbacks: Windows XP does not provide a way to check whether an object with the chosen name already exists, and the object name space is global, without regard to the object type. For instance, two applications may create an object named pipe when two distinct—and possibly different—objects are desired.
22.2 What is a handle, and how does a process obtain a handle?

22.3 Describe the management scheme of the virtual memory manager. How does the VM manager improve performance?

22.4 Describe a useful application of the no-access page facility provided in Windows XP.

22.5 The IA64 processors contain registers that can be used to address a 64-bit address space. However, Windows XP limits the address space of user programs to 8 TB, which corresponds to 43 bits’ worth. Why was this decision made?

22.6 Describe the three techniques used for communicating data in a local procedure call. What different settings are most conducive to the application of the different message-passing techniques?

22.7 What manages cache in Windows XP? How is cache managed?

22.8 What is the purpose of the Win16 execution environment? What limitations are imposed on the programs executing inside this environment? What are the protection guarantees provided between different applications executing inside the Win16 environment? What are the protection guarantees provided between an application executing inside the Windows16 environment and a 32-bit application?

22.9 Describe two user-mode processes that Windows XP provides to enable it to run programs developed for other operating systems.

22.10 How does the NTFS directory structure differ from the directory structure used in Unix operating systems?

22.11 What is a process, and how is it managed in Windows XP?

22.12 What is the fiber abstraction provided by Windows XP? How does it differ from the threads abstraction?

Bibliographical Notes


Bibliography


