

This page intentionally left blank

OPERATING SYSTEMS

INTERNAL AND DESIGN

PRINCIPLES

SEVENTH EDITION

Editorial Director: Marcia Horton

*To my brilliant and brave wife,
Antigone Tricia, who has survived
the worst horrors imaginable.*

- 7.3** Paging 321
- 7.4** Segmentation 325
- 7.5** Security Issues 326
- 7.6** Summary 330
- 7.7** Recommended Reading 330
- 7.8** Key Terms, Review Questions, and Problems 331
- 7A** Loading and Linking 334

Chapter 8 Virtual Memory 340

- 8.1**

	Operating Systems 576
13.3	eCos 579
13.4	TinyOS

20.3 Queueing Models 20-10
20.4

M.2

ONLINE R

PREFACE

This book does not pretend to be a comprehensive record; but it aims at helping to disentangle from an immense mass of material the crucial issues and cardinal decisions. Throughout I have set myself to explain faithfully and to the best of my ability.

—THE W

object-oriented design principles. This book covers the technology used in the most recent version of Windows, known as Windows 7.

- **UNIX:** A multiuser operating system, originally intended for minicomput-

This page intentionally left blank

them to real-world design choices that must be made, two operating systems have been chosen as running examples:

- **Windows:**

or need a refresher on this language. For instructors, this Web site links to course pages by professors teaching from this book and provides a number of other useful documents and links.

There is also an access-controlled Web site, referred to as **Premium Content**,

No artifact designed by man is so convenient for this kind of functional description as a digital computer. Almost the only ones of its properties that are detectable in its behavior are the organizational properties. Almost no interesting statement that one can make about on operating computer bears any particular relation to the specific nature of the hardware. A computer is an organization of elementary functional components in which, to a high approximation, only the function performed by those components is relevant to the behavior of the whole system.

THE SCIENCES OF THE ARTIFICIAL, Herbert Simon

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Describe the basic elements of a computer system and their interrelationship.
- Explain the steps taken by a processor to execute an instruction.
- Understand the concept of interrupts and how and why a processor uses interrupts.
- List and describe the levels of a typical computer memory hierarchy.
- Explain the basic characteristics of multiprocessor and multicore organizations.
- Discuss the concept of locality and analyze the performance of a multilevel memory hierarchy.
- Understand the operation of a stack and its use to support procedure call and return.

An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users. The OS also manages secondary memory and I/O (input/output) devices on behalf of its users. Accordingly, it is important to have some understanding of the underlying computer system hardware before we begin our examination of operating systems.

This chapter provides an overview of computer system hardware. In most areas, the survey is brief, as it is assumed that the reader is familiar with this subject. However, several areas are covered in some detail because of their importance to topics covered later in the book. Further topics are covered in Appendix C.

1.1 BASIC ELEMENTS

At a top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs. Thus, there are four main structural elements:

- **Processor:** Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the **central processing unit** (CPU).

- **Main memory:** Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. In contrast, the contents of disk memory are retained even when the computer system is shut down. Main memory is also referred to as *real memory* or *primary memory*.
- **I/O modules:** Move data between the computer and its external environment.

the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of

1.3 INSTRUCTION EXECUTION

A program to be executed by a processor consists of a set of instructions stored in memory. In its simplest form, instruction processing consists of two steps: The processor reads (*fetches*) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. Instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an *instruction cycle*. Using a simplified two-step description, the instruction cycle is depicted in Figure 1.2. The two steps are referred to as the *fetch stage*/*execute stage*.

halts only if the processor is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the processor is encountered.

At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the instruction to be fetched. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). For example, consider a simplified computer in which each instruction occupies one word of memory. Assume that the program counter is set to location 300.

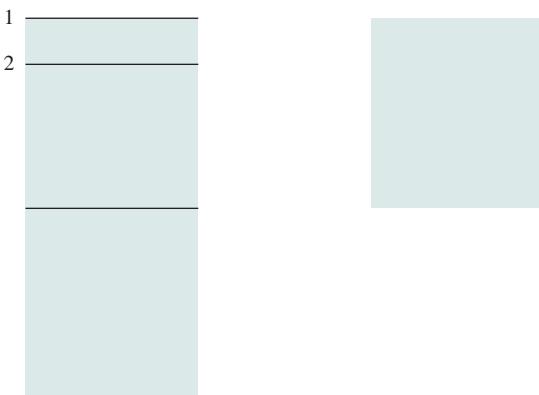
- and a memory buffer register (MBR). For simplicity, these intermediate registers are not shown.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded from memory. The remaining 12 bits (three hexadecimal digits) specify the address, which is 940.
 3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
 4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.
 5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
 6. The contents of the AC are stored in location 941.

1.4 INTERRUPTS

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. Table 1.1 lists the most common classes of interrupts.

Interrupts are provided primarily as a way to improve processor utilization. For example, most I/O devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 1.2. After each write operation, the processor must pause and remain idle

The dashed line represents the path of execution followed by the processor; that is, this line shows the sequence in which instructions are executed. Thus, after the first WRITE instruction is encountered, the user program is interrupted and execution continues with the I/O program. After the I/O program execution is complete, execution resumes in the user program immediately following the WRITE instruction.



that particular I/O device, known as an interrupt handler; and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by **X** in Figure 1.5b. Note that an interrupt can occur at any point in the main program, not just at one specific instruction.

For the user program, an interrupt suspends the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 1.6). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the OS are responsible for suspending the user program and then resuming it at the same point.

To accommodate interrupts, an *interrupt stage* is added to the instruction cycle, as shown in Figure 1.7 (compare Figure 1.2). In the interrupt stage, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program. If an interrupt is pending,

the processor suspends execution of the current program and executes an *interrupt-handler* routine. The interrupt-handler routine is generally part of the OS. Typically, this routine determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt-handler routine is completed, the processor can resume execution of the user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt and to decide on the appropriate action. Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

To appreciate the gain in efficiency, consider Figure 1.8, which is a timing dia-

assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. The more typical case, especially for a slow device such as a printer, is

Interrupt Processing

An interrupt triggers a number of events, both in the processor hardware and in software. Figure 1.10 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt, as indicated in Figure 1.7.
3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.

4.

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A *disabled interrupt*

The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more

Figure 1.15 shows the general shape of the curve that models this situation. The figure shows the average access time to a two-level memory as a function of the **hit**

memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files, and are usually visible to the programmer only in terms of files and records, as opposed to individ-

Cache Principles

Cache memory is intended to provide memory access time approaching that of the fastest memories available and at the same time support a large memory size that has the price of less expensive types of semiconductor memories. The concept is illus-

of memory that is not in the cache is read, that block is transferred to one of the slots of the cache. Because there are more blocks than slots, an individual slot cannot be uniquely and permanently dedicated to a particular block. Therefore, each slot includes a tag that identifies which particular block is currently being stored. The tag is usually some number of higher-order bits of the address and refers to all addresses that begin with that sequence of bits.

As a simple example, suppose that we have a 6-bit address and a 2-bit tag. The

addressed in dealing with virtual memory and disk cache design. They fall into the following categories:

- Cache size
- Block size
- Mapping function
- Replacement algorithm
- Write policy
- Number of cache levels

We have already dealt with the issue of **cache size**. It turns out that reasonably small caches can have a significant impact on performance. Another size issue is that of **block size**: the unit of data exchanged between cache and main memory. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality: the high probability that data in the

block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched data becomes less than the probability of reusing the data that have to be moved out of the cache to make room for the new block.

When a new block of data is read into the cache, the **mapping function** determines which cache location the block will occupy. Two constraints affect the design

the processor periodically checks the status of the I/O module until it finds that the operation is complete.

With programmed I/O, the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. The

- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:**

also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.

- a. What is the maximum directly addressable memory capacity (in bytes)?
- b. Discuss the impact on the system speed if the microprocessor bus has
 1. a 32-bit local address bus and a 16-bit local data bus, or
 2. a 16-bit local address bus and a 16-bit local data bus.
- c. How many bits are needed for the program counter and the instruction register?

Assume that one processor cycle equals one bus cycle. Now suppose that very large blocks of data are to be transferred between M and D .

- a. If programmed I/O is used and each one-word I/O transfer requires the CPU to execute two instructions, estimate the maximum I/O data transfer rate, in words

Table 1.2 Characteristics of Two-Level Memories

Memory Type	Access Time (ns)	Latency (ns)	Bandwidth (GB/s)	Power Consumption (mW)	Cost (\$/Gb)
SRAM	10–20	10–20	10–100	10–100	10–100
DRAM	100–200	100–200	10–100	10–100	10–100
RRAM	10–20	10–20	10–100	10–100	10–100
FeFET	10–20	10–20	10–100	10–100	10–100
MRAM	10–20	10–20	10–100	10–100	10–100
TCRAM	10–20	10–20	10–100	10–100	10–100
PCRAM	10–20	10–20	10–100	10–100	10–100
RRAM	10–20	10–20	10–100	10–100	10–100
FeFET	10–20	10–20	10–100	10–100	10–100
MRAM	10–20	10–20	10–100	10–100	10–100
TCRAM	10–20	10–20	10–100	10–100	10–100
PCRAM	10–20	10–20	10–100	10–100	10–100

A distinction is made in the literature between spatial locality and temporal

where

C_s = average cost per bit for the combined two-level memory

C_1 = average cost per bit of upper-level memory M1

C_2 = average cost per bit of lower-level memory M2

S_1 = size of M1

S_2 = size of M2

We would like $C_s < C_2$. Given that $C_1 > C_2$

To get at this, consider the quantity T_1/T_s , which is referred to as the *access efficiency*. It is a measure of how close average access time (T_s) is to M1 access time (T_1). From Equation (1.1),

T





Operating systems are those programs that interface the machine with the applications programs. The main function of these e program thotdynamcat

involve mounting or dismounting tapes or setting up card decks. If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence. Thus, a considerable amount of time was spent just in setting up the program to run.

This mode of operation could be termed *serial processing*, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

Simple Batch Systems

Early computers were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

memory. Once a job has been read in, the processor will encounter a branch instruction in the monitor that instructs the processor to continue execution at the start of the user program. The processor will then execute the instruc-

the 5 minutes are over and then completes 15 minutes after that. JOB3 begins after 20 minutes and completes at 30 minutes from the time it was initially submitted. The average resource utilization, throughput, and response times are shown in the uniprogramming column of Table 2.2. Device-by-device utilization is illustrated in Figure 2.6a. It is evident that there is gross underutilization for all resources when averaged over the required 30-minute time period.

Now suppose that there are four processes running simultaneously. The utilization of each device is shown in Figure 2.6a.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated personal computer or workstation. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing was developed.

Just as multiprogramming allows the processor to handle multiple batch jobs

- JOB3: 5000
- JOB4: 10,000

Initially, the monitor loads JOB1 and transfers control to it (a). Later, the

to a particular file. The contention for resources, such as printers and mass storage devices, must be handled. These and other problems, with possible solutions, will be encountered throughout this text.

2.3 MAJOR ACHIEVEMENTS

A second line of development was general-purpose time sharing. Here, the

exclusion is difficult to verify as being correct under all possible sequences of events.

- **Nondeterminate program operation:** The results of a particular program normally should depend only on the input to that program and not on the activities of other programs in a shared system. But when programs share memory, and their execution is interleaved by the processor, they may interfere with each other by overwriting common memory areas in unpredictable ways. Thus, the order in which various programs are scheduled may affect the outcome of any particular program.

- **Deadlocks:**

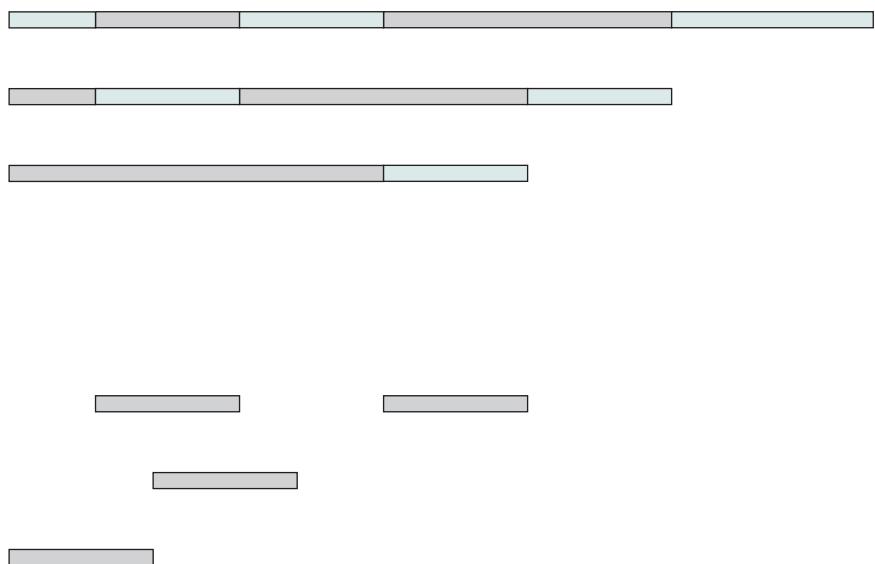
The processor hardware, together with the OS, provides the user with a “virtual processor” that has access to a virtual memory. This memory may be a

built into computers and operating systems that support a variety of protection and security mechanisms. In general, we are concerned with the problem of controlling access to computer systems and the information stored in them.

dispatcher, to pick one. A common strategy is to give each process in the queue some time in turn; this is referred to as a **round-robin** technique. In effect, the round-robin technique employs a circular queue. Another strategy is to assign priority levels to the various processes, with the scheduler selecting processes in priority order.

The long-term queue is a list of new jobs waiting to use the processor. The OS adds jobs to the system by transferring a process from the long-term queue to the short-term queue. At that time, a portion of main memory must be allocated

stack (to enable subroutine branching). A thread executes sequentially and is



individual processors and of synchronization among processors. This book discusses the scheduling and synchronization mechanisms used to provide the single-system appearance to the user. A different problem is to provide the appearance of a single system for a cluster of separate computers—a multicomputer system. In this case, we are dealing with a collection of entities (computers), each with its own main memory, secondary memory, and other I/O modules. A **distributed operating system** provides the illusion of a single main memory space and a single secondary memory space, plus other unified access facilities, such as a distributed file system. Although clusters are becoming increasingly popular, and there are many cluster products on the market, the state of the art for distributed operating systems lags that of uniprocessor and SMP operating systems. We examine such systems in Part Eight.

Another innovation in OS design is the use of object-oriented technologies. **Object-oriented design** lends discipline to the process of adding modular extensions to a small kernel. At the OS level, an object-based structure enables programmers to customize an OS without disrupting system integrity. Object orientation also eases the development of distributed tools and full-blown distributed operating systems.

2.5 VIRTUAL MACHINES

Virtual Machines and Virtualizing

Traditionally, applications have run directly on an OS on a PC or a server. Each PC or server would run only one OS at a time. Thus, the vendor had to rewrite parts of its applications for each OS/platform they would run on. An effective strategy

connections to communicate with one another, when such communication is needed. Key to the success of this approach is that the VMM provides a layer between software environments and the underlying hardware and host OS that is programmable, transparent to the software above it, and makes efficient use of the hardware below it.

Virtual Machine Architecture²

Recall from Section 2.1 (see Figure 2.1) the discussion of the application programming interface, the application binary interface, and the instruction set architecture. Let us use these interface concepts to clarify the meaning of *machine* in the term *virtual machine*.

process, the machine on which it executes

SYSTEM VIRTUAL MACHINE In a system VM, virtualizing software translates the ISA used by one hardware platform to that of another. Note in Figure 2.14a that the virtualizing software in the process VM approach makes use of the services of the host OS, while in the system VM approach there is logically no separate host

2.6 OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Symmetric Multiprocessor OS Considerations

In an SMP system, the kernel can execute on any processor, and typically each

- **Reliability and fault tolerance:** The OS should provide graceful degradation in the face of processor failure. The scheduler and other portions of the OS

GCD does not help the developer decide how to break up a task or application into

Windows 7 and Windows Server 2008 R2. Despite the difference in naming, the

Architecture

Figure 2.15 illustrates the overall structure of Windows 7; all releases of Windows

- It provides a suitable base for distributed computing. Typically, distributed

class is based on a base class which specifies virtual methods that support

ess A wishes to synchronize with process B, it could create a named event object and pass the name of the event to B. Process B could then open and use that event object. However, if A simply wished to use the event to synchronize two threads within itself, it would create an unnamed event object, because there is no need for other processes to be able to use that event.

There are two categories of objects used by Windows for synchronizing the use of the processor:

- **Dispatcher objects:** The subset of Executive objects which threads can wait on to control the dispatching and synchronization of thread-based system operations. These are described

- **Engineering improvements:** The performance of hundreds of key scenarios, such as opening a file from the GUI, are tracked and continuously characterized to identify and fix problems. The system is now built in layers which can be separately tested, improving modularity and reducing complexity.
- **Performance improvements:** The amount of memory required has been reduced, both for clients and servers. The VMM is more aggressive about

2.8 TRADITIONAL UNIX SYSTEMS

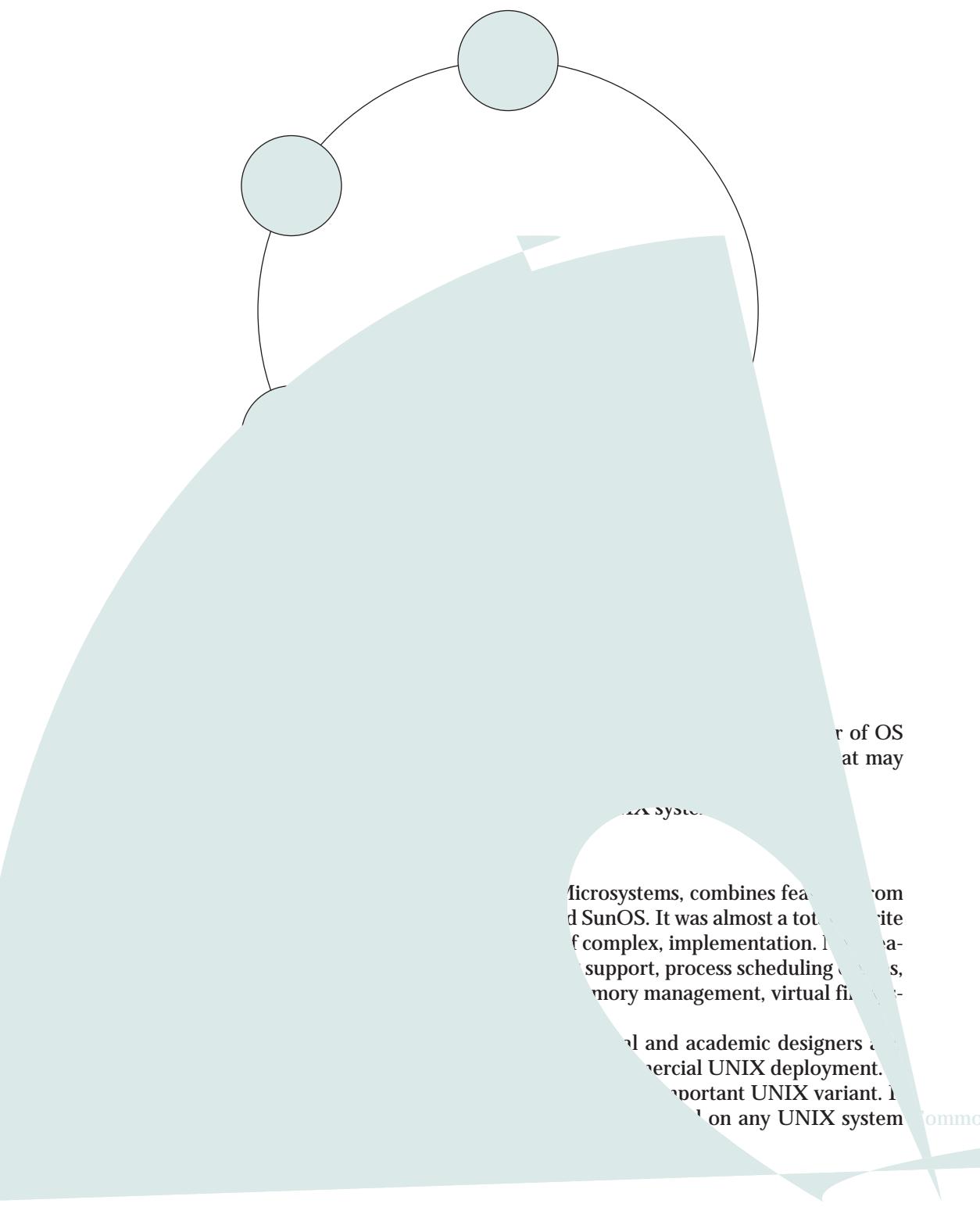
History

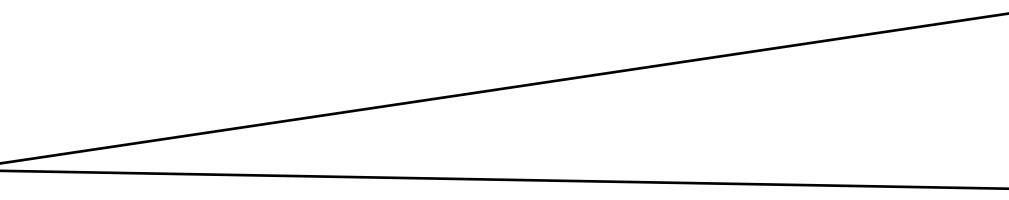
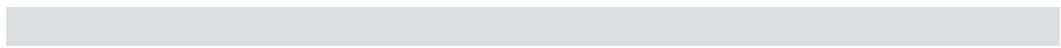
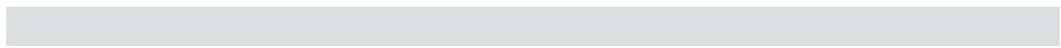
The history of UNIX is an oft-told tale and will not be repeated in great detail here. Instead, we provide a brief summary.

UNIX was initially developed at Bell Labs and became operational on a shared computer system at BELL Labs in 1971. It was designed to be a multi-user, multi-tasking operating system.

Description

made about a traditional UNIX system. It is designed to run on a single processor





- **File systems:** Provides a global, hierarchical namespace for files, directories, and other file related objects and provides file system functions.
- **Network protocols:** Supports the Sockets interface to users for the TCP/IP protocol suite.
- **Character device drivers:**

other contexts but cannot affect them. All other contexts provide complete isolation: Processes from one context can neither see nor interact with processes from another context. This provides the ability to run similar contexts on the same computer without any interaction possible at the application level. Thus, each virtual

[MUKH96] provides a good discussion of OS design issues for SMPs.

- 2.6** List and briefly explain five storage management responsibilities of a typical OS.
- 2.7** Explain the distinction between a real address and a virtual address.
- 2.8** Describe the round-robin scheduling technique.
- 2.9** Explain the difference between a monolithic kernel and a microkernel.
- 2.10** What is multithreading?
- 2.11** List the key design issues for an SMP operating system.

Problems

- 2.1** Suppose that we have a multiprogrammed computer in which each job has identical characteristics. In one computation period, T , for a job, half the time is spent in I/O

drives to complete execution. Assume that each job starts running with only three tape drives for a long period before requiring the fourth tape drive for a short period toward the end of its operation. Also assume an endless supply of such jobs.

- a. Assume the scheduler in the OS will not start a job unless there are four tape

3.1

$$\Xi = \{ \cdot \} \cup \mathbf{A}_{\mathcal{X}} \cup \mathbf{A}_{\mathcal{Y}} \cup \mathcal{V}$$

register. Over time, the program counter may refer to code in different programs that are part of different processes. From the point of view of an individual program, its execution involves a sequence of instructions within that program.

We can characterize the behavior of an individual process by listing the sequence of instructions that execute for that process. Such a listing is referred to as a **trace** of the process. We can characterize behavior of the processor by showing how the traces of the various processes are interleaved.

Let us consider a very simple example. Figure 3.2 shows a memory layout of three processes. To simplify the discussion, we assume no use of virtual memory; thus all three processes are represented by programs that are fully loaded in main memory. In addition, there is a small **dispatcher** program that switches the processor from one process to another. Figure 3.3 shows the traces of each of the processes during the early part of their execution. The first 12 instructions executed in processes A and C are shown. Process B executes four instructions, and we assume that the fourth instruction invokes an I/O operation for which the

Running state to the Not Running state, and one of the other processes moves to the Running state.

From this simple model, we can already begin to appreciate some of the design elements of the OS. Each process must be represented in some way so that the OS can keep track of it. That is, there must be some information relating to each proc-

the queue may consist of a linked list of data blocks, in which each block represents one process; we will explore this latter implementation subsequently.

We can describe the behavior of the dispatcher in terms of this queueing diagram. A process that is interrupted is transferred to the queue of waiting processes. Alternatively, if the process has completed or aborted, it is discarded (exits the system). In either case, the dispatcher takes another process from the queue to execute.

The Creation and Termination of Processes

Before refining our simple two-state model, it will be useful to discuss the creation and termination of processes; ultimately, and regardless of the model of process behavior that is used, the life of a process is bounded by its creation and termination.

PROCESS CREATION When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address space in main memory to the process. We describe these data structures in Section 3.3. These actions constitute the creation of a new process.

Four common events lead to the creation of a process, as indicated in Table 3.1. In a batch environment, a process is created in response to the submission of a job.

Traditionally, the OS created all processes in a way that was transparent to the user or application program, and this is still commonly found with many contem-

we have added two additional states that will prove useful. The five states in this new diagram are:

- **Running:** The process that is currently being executed. For this chapter, we will assume a computer with a single processor, so at most one process at a time can be in this state.
- **Ready:** A process that is prepared to execute when given the opportunity.
- **Blocked/Waiting:**⁴ A process that cannot execute until some event occurs,

not in main memory. That is, the code of the program to be executed is not in main memory, and no space has been allocated for the data associated with that program. While the process is in the New state, the program remains in secondary storage, typically disk storage.⁵

Similarly, a process exits a system in two stages. First, a process is terminated

say that the OS has

queue. In the absence of any priority scheme, this can be a simple first-in-first-out

But this line of reasoning presents a difficulty. All of the processes that have been suspended were in the Blocked state at the time of suspension. It clearly would not do any good to bring a blocked process back into main memory, because it is still not ready for execution. Recognize, however, that each process in the Suspend state was originally blocked on a particular event. When that event occurs, the process is not blocked and is potentially available for execution.

Therefore, we need to rethink this aspect of the design. There are two independent concepts here: whether a process is waiting on an event (blocked or not)

- **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
- **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.

Before looking at a state transition diagram that encompasses the two new suspend states, one other point should be mentioned. The discussion so far has assumed that virtual memory is not in use and that a process is either all in main memory or all out of main memory. With a virtual memory scheme, it is possible to execute a process that is only partially in main memory. If reference is made to a process address that is not in main memory, then the appropriate portion of the process can be brought in. The use of virtual memory would appear to eliminate the need for explicit swapping, because any desired address in any desired process can be moved in or out of main memory by the memory management hardware of the processor. However, as we shall see in Chapter 8, the performance of a virtual memory system can collapse if there is a sufficiently large number of active processes, all of which are partially in main memory. Therefore, even in a virtual memory system, the OS will need to swap out processes explicitly and completely from time to time in the interests of performance.

Let us look now, in Figure 3.9b, at the state transition model that we have developed. (The dashed lines in the figure indicate possible but not necessary transitions.) Important new transitions are the following:

- **Blocked → Blocked/Suspend:** If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked. This transition can be made even if there are ready processes

1 • 1 Tf1.0072 0 TD63 Tc0 Twocessor. end: If 26/oc As swapped re indsp

Several other transitions that are worth considering are the following:

- **New → Ready/Suspend and New → Ready:** When a new process is created, it can either be added to the Ready queue or the Ready/Suspend queue. In either case, the OS must create a process control block and allocate an address space to the process. It might be preferable for the OS to perform these housekeeping duties at an early time, so that it can maintain a large pool of processes that are not blocked. With this strategy, there would often be insufficient room in main memory for a new process; hence the use of the (New → Ready/Suspend) transition. On the other hand, we could argue that a just-in-time philosophy of creating processes as late as possible reduces OS overhead and allows that OS to perform the process-creation duties at a time when the system is clogged with blocked processes anyway.
- **Blocked/Suspend → Blocked:** Inclusion of this transition may seem to be poor design. After all, if a process is not ready to execute and is not already in main memory, what is the point of bringing it in? But consider the following scenario: A process terminates, freeing up some main memory. There is a process in the (Blocked/Suspend) queue with a higher priority than any of the processes in the (Ready/Suspend) queue and the OS has reason to believe that

3. The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution.
4. The process may not be removed from this state until the agent explicitly orders the removal.

Table 3.3 lists some reasons for the suspension of a process. One reason that we have discussed is to provide memory space either to bring in a Ready/Suspended process or to increase the memory allocated to other Ready processes. The OS may have other motivations for suspending a process. For example, an auditing or tracing process may be employed to monitor activity on the system; the process may be used to record the level of utilization of various resources (processor, memory, channels) and the rate of progress of the user processes in the system. The OS, under operator control, may turn this process on and off from time to time. If the OS detects or s for, an 7ec0 im, it may suspend a process. One example of this is deadlock, which is discussed in Chapter 6. Another example, a7ec0 im is detected on a communications line, and the operator has the OS suspend the process that is using the line while some tests are run.

Another set of reasons concerns the 0132.2E of nd ippe actihe uses. For example, if t suspendin t its executio,r exaining endmodifysing the progaie erdatae, andgreumsing execution t

- Any protection attributes of blocks of main or virtual memory, such as which processes may access certain shared memory regions
-

files are managed on behalf of processes, so there must be some reference to these resources, directly or indirectly, in the process tables. The files referred to in the file tables are accessible via an I/O device and will, at some times, be in main or virtual memory. The tables themselves must be accessible by the OS and therefore are subject to memory management.

Second, how does the OS know to create the tables in the first place? Clearly, the OS must have some knowledge of the basic environment, such as how much main memory exists, what are the I/O devices and what are their identifiers, and so on. This is an issue of configuration. That is, when the OS is initialized, it must have access to some configuration data that define the basic environment, and these data must be created outside the OS, with human assistance or by some autoconfiguration software.

Process Control Structures

Consider what the OS must know if it is to manage and control a process. First, it must know where the process is located; second, it must know the attributes of the process that are necessary for its management (e.g., process ID and process state).

PROCESS LOCATION Before we can deal with the questions of where a process is located or what its attributes are, we need to address an even more fundamental question: What is the physical manifestation of a process? At a minimum, a process must include a program or set of programs to be executed. Associated with these programs is a set of data locations for local and global variables and any defined constants. Thus, a process will consist of at least sufficient memory to hold the programs and data of that process. In addition, the execution of a program typically involves a stack (see Appendix P) that is used to keep track of procedure calls and parameter passing between procedures. Finally, each process has associated with it a number of attributes that are used by the OS for process control. Typically, the collection of attributes is referred to as a *process control block*.⁷ We can refer to this collection of program, data, stack, and attributes as the **process image** (Table 3.4).

The location of a process image will depend on the memory management system. The process image will be located in memory, and the memory will be mapped to the process's address space.

70.12.CI.S.BT Tf 24.00810 368 608.

Table 3.5 Typical Elements of a Process Control Block

The following table lists typical elements of a process control block.

Table 3.5 is located on page 130 of the book.

access to these tables is not difficult. Each process is equipped with a unique ID, and this can be used as an index into a table of pointers to the process control blocks. The difficulty is not access but rather protection. Two problems present themselves:

for user mode. When an interrupt occurs, the processor clears most of the bits in the psr, including the cpl field. This automatically sets the cpl to level 0. At the end of the interrupt-handling routine, the final instruction that is executed is irt (interrupt return). This instruction causes the processor to restore the psr of the interrupted program, which restores the privilege level of that program. A similar sequence occurs when an application places a system call. For the Itanium, an application places a

Process Switching

On the face of it, the function of process switching would seem to be straightforward. At some time, a running process is interrupted and the OS assigns another process to the Running state and turns control over to that process. However, several design issues are raised. First, what events trigger a process switch? Another issue is that

3.5 EXECUTION OF THE OPERATING SYSTEM

In Chapter 2, we pointed out two intriguing facts about operating systems:

- The OS functions in the same way as ordinary computer software in the sense that the OS is a set of programs executed by the processor.

user process. Alternatively, the OS can complete the function of saving the environment of the process and proceed to schedule and dispatch another process. Whether this happens depends on the reason for the interruption and the circumstances at the time.

In any case, the key point here is that the concept of process is considered to apply only to user programs. The operating system code is executed as a separate entity that operates in privileged mode.

Execution within User Processes

An alternative that is common with operating systems on smaller computers (PCs, workstations) is to execute virtually all OS software in the context of a user process. The view is that the OS is primarily a collection of routines that the user calls to perform various functions, executed within the environment of the user's process.

Operating system code and data are in the shared address space and are shared by all user processes.

level and be interleaved with other processes under dispatcher control. Finally,

out there. At the serious end are individuals who are attempting to read privileged data, perform unauthorized modifications to data, or disrupt the system.

The objective of the intruder is to gain access to a system or to increase the range of privileges accessible on a system. Most initial attacks use system or soft-

An IDS comprises three logical components:

- **Sensors:** Sensors are responsible for collecting data. The input for a sensor may be any part of a system that could contain evidence of an intrusion. Types of input to a sensor include network packets, log files, and system call traces. Sensors collect and forward this information to the analyzer.
- **Analyzers:** Analyzers receive input from one or more sensors or from other analyzers. The analyzer is responsible for determining if an intrusion has occurred. The output of this component is an indication that an intrusion has occurred. The output may include evidence supporting the conclusion that an intrusion occurred. The analyzer may provide guidance about what actions to take as a result of the intrusion.
- **User interface:** The user interface to an IDS enables a user to view output

- [BELL94] lists the following design goals for a firewall:
1. All traffic from inside to outside, and vice versa, must pass through the fire-

(based on figure in [BACH86]). This figure is similar to Figure 3.9b, with the two UNIX sleeping states corresponding to the two blocked states. The differences are as follows:

- UNIX employs two Running states to indicate whether the process is executing in user mode or kernel mode.
- A distinction is made between the two states: (Ready to Run, in Memory) and (Preempted). These are essentially the same state, as indicated by the dotted line joining them. The distinction is made to emphasize the way in which the

to each sharing process that the shared memory region is in its address space. When a process is not running, the processor status information is stored in the **register context** area.

The **system-level context** contains the remaining information that the OS needs to manage the process. It consists of a static part, which is fixed in size and stays with a process throughout its lifetime, and a dynamic part, which varies in size through the life of the process. One element of the static part is the process

3.8

- Transfer control to the child process. The child process begins executing at the same point in the code as the parent, namely at the return from the fork call.
- Transfer control to another process. Both parent and child are left in the Ready to Run state.

It is perhaps difficult to visualize this method of process creation because both parent and child are executing the same passage of code. The difference is this: When the return from the fork occurs, the return parameter is tested. If the value is zero, then this is the child process, and a branch can be executed to the appropriate user program to continue execution. If the value is nonzero, then this is the parent process, and the main line of execution can continue.

3.8 SUMMARY

The most fundamental concept in a modern OS is the process. The principal func-

The four modes are as follows:

- **Kernel:** Executes the kernel of the VMS operating system, which includes memory management, interrupt handling, and I/O operations
- **Executive:**

4.1 Processes and Tp004-5225

distinguish the two characteristics, the unit of dispatching is usually referred to as a thread or **lightweight process**, while the unit of resource ownership is usually referred to as a **process** or **task**.¹

Multithreading

Multithreading refers to the ability of an OS to support multiple, concurrent paths

periodic backup and that schedules itself directly with the OS; there is no need for fancy code in the main program to provide for time checks or to coordinate input and output.

- **Speed of execution:** A multithreaded process can compute one batch of data while reading the next batch from a device. On a multiprocessor system, multiple threads from the same process may be able to execute simultaneously. Thus, even though one thread may be blocked for an I/O operation to read in



While this multiprogramming can result in a significant speedup of the application, there are applications that would benefit from the ability to execute portions of code simultaneously.

There are ways to work around these two problems. For example, both prob-

and distribution of work to multiple processors and cache coherence overhead. This

4.4 WINDOWS 7 THREAD AND SMP MANAGEMENT

Windows process design is driven by the need to provide support for a variety of OS environments. Processes supported by different OS environments differ in a number of ways, including the following:

- How processes are named
- Whether threads are provided within processes
- How processes are represented
- How process resources are protected

token, called the primary token of the process. When a user first logs on, Windows

Multithreading

Windows supports concurrency among processes because threads in different

processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice.

- **Running:** Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher-priority thread, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the Ready state.

1) it ~~Waits for memory~~ (waits for memory synchronization) 3) it

- **WTransition**

4.5 SOLARIS THREAD AND SMP MANAGEMENT

Solaris implements multilevel thread support designed to provide considerable flexibility in exploiting processor resources.

The solution in Solaris can be summarized as follows:

1. Solaris employs a set of kernel threads to handle interrupts. As with any kernel thread, an interrupt thread has its own identifier, priority, context, and stack.
2. The kernel controls access to data structures and synchronizes among inter-

- **Stopped:**

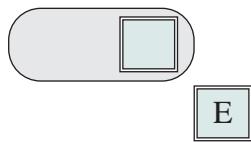
4.7 MAC OS X GRAND CENTRAL DISPATCH

As was mentioned in Chapter 2, Mac OS X Grand Central Dispatch (GCD) pro-

Blocks enable the programmer to encapsulate complex functions, together with their arguments and data, so that they can easily be referenced and passed around in a program, much like a variable.⁹ Symbolically:

$$\boxed{F} = F + \boxed{}$$

already running. This allows rapid response without the expense of polling or “parking a thread” on the event source.



All functions in GCD begin with `dispatch_`. The outer `dispatch_async()` call puts a task on a global concurrent queue. This tells the OS that the block can be assigned to a separate concurrent queue, off the main queue, and executed in parallel. Therefore, the main thread of execution is not delayed. When the `analyze` function is complete, the inner `dispatch_async()` call is encountered. This directs the OS to put the following block of code at the end of the main queue, to be executed when it reaches the head of the queue. So, with very little work on the part of the programmer, the desired requirement is met.

4.8 SUMMARY

Some operating systems distinguish the concepts of process and thread, the for-

Review Questions

4.1

1. A **vector** is a quantity that has both magnitude and direction.

2. A **scalar** is a quantity that has only magnitude.

3. A **position vector** is a vector drawn from the origin to a point in space.

4. A **displacement vector** is a vector drawn from the initial position to the final position of an object.

5. A **unit vector** is a vector with a magnitude of one.

6. A **zero vector** is a vector with a magnitude of zero.

7. A **vector component** is a vector drawn perpendicular to a reference vector.

8. A **vector projection** is the component of a vector in the direction of another vector.

9. A **vector dot product** is the scalar product of two vectors.

10. A **vector cross product** is the vector product of two vectors.


```
void count_positives(list l)
{
    list p;
    for (p = l; p; p = p -> next)
        if (p -> val > 0.0)
            ++global_positives;
}
```

Now consider the case in which thread A performs

```
count_positives(<list containing only negative values>);
```

while thread B performs

```
++global_positives;
```

- a. What does the function do?
 - b. The C language only addresses single-threaded execution. Does the use of two parallel threads create any problems or potential problems?
- 4.8** But some existing optimizing compilers (including gcc, which tends to be relatively conservative) will “optimize” `count_positives` to something similar to

```
void count_positives(list l)
{
    list p;
    register int r;
```

```
        return NULL;
    }
    int main(void) {
        pthread_t mythread;
        int i;
        if ( pthread_create( &mythread, NULL, thread_function,
            NULL) ) {
            printf(ldquo;error creating thread.");
            abort();
        }
        for ( i=0; i<20; i++) {
            myglobal=myglobal+1;
            printf("o");
            fflush(stdout);
            sleep(1);
        }
        if ( pthread_join ( mythread, NULL ) ) {
            printf("error joining thread.");
            abort();
        }
        printf("\nmyglobal equals %d\n",myglobal);
        exit(0);
    }
```

In `main()` we first declare a variable called `mythread`,

•

5.1 PRINCIPLES OF CONCURRENCY

Process Interaction

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence. Table 5.2 lists three possible degrees of awareness plus the consequences of each:

- **Processes unaware of each other:** These are independent processes that are not intended to work together. The best example of this situation is the multiprogramming of multiple independent processes. These can either be batch jobs or interactive sessions or a mixture. Although the processes are not working together, the OS needs to be concerned about

COMPETITION AMONG PROCESSES FOR RESOURCES

express the requirement for mutual exclusion in some fashion, such as locking a resource prior to its use. Any solution will involve some support from the OS, such as the provision of the locking facility. Figure 5.1 illustrates the mutual exclusion mechanism in abstract terms. There are n

must also update the other to maintain the relationship. Now consider the following two processes:

Interrupt Disabling

In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts. A process can then enforce mutual exclusion in the following way (compare Figure 5.1):

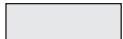
```
    (true) {  
/* disable interrupts */;  
/* critical section */;  
/* enable interrupts */;  
/* remainder */;  
}
```

Because the critical section cannot be interrupted, mutual exclusion is guaranteed. The price of this approach, however, is high. The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes. Another problem is that this approach will not work in a multiprocessor architecture. When the computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion. In **ng al**) for **I**

```
    compare_and_swap ( *word, testval, newval)  
{
```

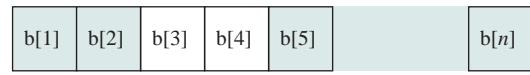
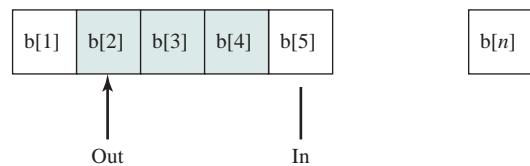

must be the one to unlock it (sets the value to 1). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.⁵

Processor



A fix for the problem is to introduce an auxiliary variable that can be set in the

The producer and consumer functions can be expressed as follows (variable *in* and *out*)



In

processes blocked waiting for monitor availability. Once a process is in the monitor, it may temporarily block itself on condition x by issuing `cwait(x)`; it is then placed in a queue of processes waiting to reenter the monitor when the condition changes, and resume execution at the point in its program following the

the process signals the *notempty* condition. A similar description can be made of the consumer function.

This example points out the division of responsibility with monitors compared to semaphores. In the case of monitors, the monitor construct itself enforces mutual exclusion: It is not possible for both a producer and a consumer simultaneously to access the buffer. However, the programmer must place the appropriate `cwait` and `csignal`

enters the monitor before activation. Otherwise, the condition under which the process was activated could change. For example, in Figure 5.16, when a `csignal(notempty)` is issued, a process from the `notempty` queue must be activated before a new consumer enters the monitor. Another example: a producer process may append a character to an empty buffer and then fail before signaling; any processes in the `notempty`

With the rule that a process is notified rather than forcibly reactivated, it is possible to add a `cbroadcast` primitive to the repertoire. The broadcast causes all processes waiting on a condition to be placed in a Ready state. This is convenient in situations where a process does not know how many other processes should be reactivated. For example, in the producer/consumer program, suppose that both the `append` and the `take` functions can apply to variable length blocks of characters. In that case, if a producer adds a block of characters to the buffer, it need not know how many characters each waiting consumer is prepared to consume. It simply issues a `cbroadcast` and all waiting processes are alerted to try again.

In addition, a broadcast can be used when a process would have difficulty figuring out precisely which other process to reactivate. A good example is a memory manager. The manager has j bytes free; a process frees up an additional k bytes, but it does not know which waiting process can proceed with a total of $k + j$ bytes. Hence it uses broadcast, and all processes check for themselves if there is enough memory free.

An advantage of Lampson/Redell monitors over Hoare monitors is that the Lampson/Redell approach is less prone to error. In the Lampson/Redell approach, because each procedure checks the monitor variable after being signaled, with the use of the `while` construct, a process can signal or broadcast incorrectly without evant variable and, if the desired condition is not met, continue to wait. Another advantage is a more modular approach to program construction. For example, consider the implementation of a buffer allocator. There are two levels of conditions to be satisfied for cooperating sequential processes:

1.

5.5 MESSAGE PASSING

When processes interact with one another, two fundamental requirements must be satisfied: synchronization and communication. Processes need to be synchronized to enforce mutual exclusion; cooperating processes may need to exchange information. One approach to providing both of these functions is message passing. Message passing has the further advantage that it lends itself to implementation in distributed systems as well as in shared-memory multiprocessor and uniprocessor systems.

Message-passing systems come in many forms. In this section, we provide a general introduction that discusses features typically found in such systems. The actual function of message passing is normally provided in the form of a pair of primitives:

```
send (destination, message)  
receive (source, message)
```

receiv/Fr03rde a

Synchronization

the association of a sender to a mailbox may occur dynamically. Primitives such as `connect` and

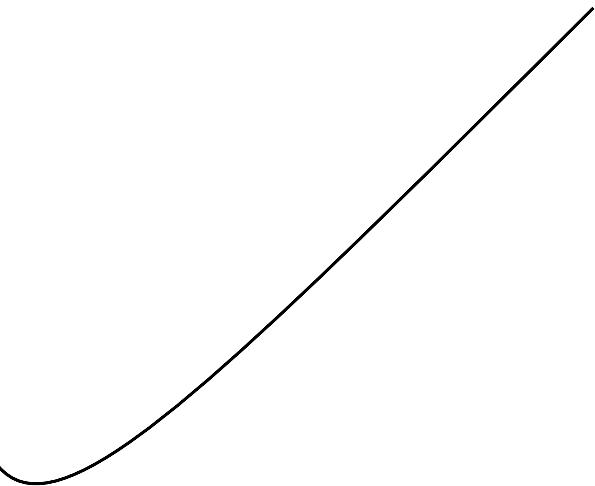
such as a pointer field so that a linked list of messages can be created; a sequence number, to keep track of the number and order of messages passed between source and destination; and a priority field.

Queueing Discipline

The simplest queueing discipline is first-in-first-out, but this may not be sufficient if some messages are more urgent than others. An alternative is to allow the specifying of message priority, on the basis of message type or by designation by the sender. Another alternative is to allow the receiver to inspect the message queue and select which message to receive next.

Mutual Exclusion

Figure 5.20 shows one way in which message passing can be used to enforce mutual exclusion (compare Figures 5.1, 5.2, and 5.6). We assume the use of the blocking receive primitive and the nonblocking send primitive. A set of concurrent pro-



The mailbox is initialized to contain a single message with null content. A process

writers, the following semaphores and variables are added to the ones already defined:

- A semaphore `rsem` that inhibits all readers while there is at least one writer

- 5.9** What is the difference between binary and general semaphores?
5.10 What is the difference between strong and weak general semaphores?

5.7 ~~for(j=0;ch!=0;j++) { if(ch>='A' & & ch<='Z') { if(ch=='A' || ch=='E' || ch=='I' || ch=='O' || ch=='U') { if(ch=='A') s[0]=ch; else if(ch=='E') s[1]=ch; else if(ch=='I') s[2]=ch; else if(ch=='O') s[3]=ch; else if(ch=='U') s[4]=ch; } else { if(ch=='B' || ch=='C' || ch=='D' || ch=='F' || ch=='G' || ch=='H' || ch=='J' || ch=='K' || ch=='L' || ch=='M' || ch=='N' || ch=='P' || ch=='Q' || ch=='R' || ch=='S' || ch=='V' || ch=='W' || ch=='X' || ch=='Y') { if(ch=='B') s[0]=ch; else if(ch=='C') s[1]=ch; else if(ch=='D') s[2]=ch; else if(ch=='F') s[3]=ch; else if(ch=='G') s[4]=ch; else if(ch=='H') s[5]=ch; else if(ch=='J') s[6]=ch; else if(ch=='K') s[7]=ch; else if(ch=='L') s[8]=ch; else if(ch=='M') s[9]=ch; else if(ch=='N') s[10]=ch; else if(ch=='P') s[11]=ch; else if(ch=='Q') s[12]=ch; else if(ch=='R') s[13]=ch; else if(ch=='S') s[14]=ch; else if(ch=='V') s[15]=ch; else if(ch=='W') s[16]=ch; else if(ch=='X') s[17]=ch; else if(ch=='Y') s[18]=ch; } } } } }~~

Compare this set of definitions with that of Figure 5.3. Note one difference: With the preceding definition, a semaphore can never take on a negative value. Is there

5.14 Now consider this correct solution to the preceding problem:

$$1 \rightarrow \mathbf{a}$$

```

14     (waiting > 0 && !must_wait)           /* If there are others waiting */
15         semSignal(block);                /* and we don't yet have 3 active, */
16                                         /* unblock a waiting process */
17     & semSignal(mutex);                 /* otherwise open the mutual exclusion */
18
19 /* critical section */
20
21 semWait(mutex);
22 --active;
23 (active == 0)
24 must_wait = false;
25 (waiting == 0 && !must_wait)
26 semSignal(block);
27
28 & semSignal(mutex);                 /* Enter mutual exclusion */
                                         /* and update the active count */
                                         /* If last one to leave? */
                                         /* set up to let new processes enter */
                                         /* If there are others waiting */
                                         /* and we don't have 3 active, */
                                         /* unblock a waiting process */
                                         /* otherwise open the mutual exclusion */

```

- a. Explain how this program works and why it is correct.
 - b. Does this solution differ from the preceding one in terms of the number of processes that can be unblocked at a time? Explain.
 - c. This program is an example of a general design pattern that is a uniform way to implement solutions to many concurrency problems using semaphores. It has been referred to as the **Pass The Baton** pattern. Describe the pattern.
- 5.16** It should be possible to implement general semaphores using binary semaphores. We can use the operations `semWaitB` and `semSignalB` and two binary semaphores, `rots561 Tf105556 0 TD[()2.8(and)]TJ/F42 1 T11f1.2 Onal(mfrots561`

this conjecture by publishing an algorithm using three weak semaphores. The behavior of the algorithm can be described as follows: If one or several process are waiting in a `semWait(S)` operation and another process is executing `semSignal(S)`, the value of the semaphore S is not modified and one of the waiting processes is unblocked independently of `semWait(S)`. Apart from the three semaphores, the algorithm uses two nonnegative integer variables as counters of the number of processes in certain sections of the algorithm. Thus, semaphores A and B are initialized to 1, while semaphore M and counters NA and NM are initialized to 0. The mutual exclusion semaphore B pro-



All deadlocks involve conflicting needs for resources by two or more processes. A common example is the traffic deadlock. Figure 6.1a shows a situation in which four cars have arrived at a four-way stop intersection at approximately the same time. The four quadrants of the intersection are the resources over which control is needed. In particular, if all four cars wish to go straight through the intersection, the resource requirements are as follows:

- Car 1, traveling north, needs quadrants a and b.
- Car 2 needs quadrants b and c.
- Car 3 needs quadrants c and d.
- Car 4 needs quadrants d and a.

The rule of the road in the United States is that a car at a four-way stop should

may compete for the same resource, a higher-dimensional diagram would be required. The principles concerning fatal regions and deadlock would remain the same.

Reusable Resources

could be in use for a considerable period of time, even years, before the deadlock actually occurs.

There is no single effective strategy that can deal with all types of deadlock. Table 6.1 summarizes the key elements of the most important approaches that have been developed: prevention, avoidance, and detection. We examine each of these in turn, after first introducing resource allocation graphs and then discussing the conditions for deadlock.

Resource Allocation Graphs

A useful tool in characterizing the allocation of resources to processes is the **resource allocation graph**, introduced by Holt [HOLT72]. The resource allocation graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node. A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted (Figure 6.5a). Within a resource node, a dot is shown for each instance of that resource. Examples of resource types that may have

has been assigned one unit of that resource. A graph edge directed from a consumable resource node dot to a process indicates that the process is the producer of that resource.

Figure 6.5c shows an example deadlock. There is only one unit each of resources Ra and Rb. Process P1 holds Rb and requests Ra, while P2 holds Ra but requests Rb. Figure 6.5d has the same topology as Figure 6.5c, but there is no deadlock because multiple units of each resource are available.

The fourth condition is, actually, a potential consequence of the first three.

Mutual Exclusion

In general, the first of the four listed conditions cannot be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by

6.3 DEADLOCK AVOIDANCE

An approach to solving the deadlock problem that differs subtly from deadlock prevention is deadlock avoidance.² In **deadlock prevention**, we constrain resource requests to prevent at least one of the four conditions of deadlock. This is either

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

	R1	R2	R3	R4	R5
P1	0	1	0	0	1

occur when an application includes concurrent threads of execution. Accordingly, this problem is a standard test case for evaluating approaches to synchronization.

Pipes, messages, and shared memory can be used to communicate data between processes, whereas semaphores and signals are used to trigger actions by other processes.

Pipes

One of the most significant contributions of UNIX to the development of operating systems is the pipe. Inspired by the concept of coroutines [RITC84], a pipe is a circular buffer allowing two processes to communicate on the producer-consumer model.

A semaphore consists of the following elements:

6.8 LINUX KERNEL CONCURRENCY MECHANISMS

Linux includes all of the concurrency mechanisms found in other UNIX systems, such as SVR4, including pipes, messages, shared memory, and signals. In addition, Linux 2.6 includes a rich set of concurrency mechanisms specifically intended for use when a thread is executing in kernel mode. That is, these are mechanisms used within the kernel to provide concurrency in the execution of kernel code. This section examines the Linux kernel concurrency mechanisms.

Atomic Operations

are allowed on this data type. [LOVE04] lists the following advantages for these restrictions:

1.

spinlock for a thread that intends to update the data structure. Each reader-writer

Linux as -EINTR. This alerts the thread that the invoked semaphore function

Barriers

In some architectures, compilers and/or the processor hardware may reorder memory accesses in source code to optimize performance. These reorderings are done to optimize the use of the instruction pipeline in the processor. The reordering algorithms contain checks to ensure that data dependencies are not violated. For example, the code:

```
a = 1;  
b = 1;
```

may be reordered so that memory location b

Given a square matrix A , we can find a matrix A^{-1} such that $A \cdot A^{-1} = A^{-1} \cdot A = I$.

The first five object types in the table are specifically designed to support synchronization. The remaining object types have other uses but also may be used for synchronization.

Each dispatcher object instance can be in either a signaled or unsignaled state. A thread can be blocked on an object in an unsignaled state; the thread is released when the object enters the signaled state. The mechanism is straightforward: A thread issues a wait request to the Windows Executive, using the handle of the synchronization object. When an object enters the signaled state, the Windows Executive releases one or all of the thread objects that are waiting on that dispatcher object.

The event object

The dispatcher object is only allocated as a last resort. Most critical sections are needed for correctness, but in practice are rarely contended. By lazily allocating the

298

$$\mathbf{A}_{\perp} \mathbf{v} = \mathbf{v} - (\mathbf{A}_{\parallel} \mathbf{v}) \quad \mathbf{A}_{\perp} \mathbf{A}_{\parallel} = \mathbf{A}_{\parallel} \mathbf{A}_{\perp} = 0$$

6.13 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

blocks of equal size. These blocks are buffered on a disk using a floating boundary between the input and the output buffers, depending on the speed of the processes. The communication primitives used ensure that the following resource constraint is satisfied:

$$i + o \leq \max$$

where

\max = maximum number of blocks on disk

i = number of input blocks on disk

o = number of output blocks on disk

The following is known about the processes:

1. As long as the environment supplies data, process I will eventually-

6.11

	x_1	x_2	x_3	x_4
r_1	1	0	0	0
r_2	0	1	0	0
r_3	0	0	1	0
r_4	0	0	0	1

I cannot guarantee that I carry all the facts in my mind. Intense mental concentration has a curious way of blotting out what has passed. Each of my cases displaces the last, and Mlle. Carère has blurred my recollection of Baskerville Hall. Tomorrow some other little problem may be submitted to my notice which will in turn dispossess the fair French lady and the infamous Upwood.

—THE HOUND OF THE BASKERVILLES,
Arthur Conan Doyle.

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

-

that is not present, the user's program must load that module into the program's partition, overlaying whatever programs or data are there.

- Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. In our example, there may be a program whose length is less than 2 Mbytes; yet it occupies an 8-Mbyte partition whenever it is swapped in. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as **internal fragmentation**.

Both of these problems can be lessened, though not solved, by using unequal-size partitions (Figure 7.2b). In this example, programs as large as 16 Mbytes can be accommodated without overlays. Partitions smaller than 8 Mbytes allow smaller

loaded into that partition. Because all partitions are of equal size, it does not matter

consider other factors, such as priority, and a preference for swapping out blocked processes versus ready processes.

The use of unequal-size partitions provides a degree of flexibility to fixed partitioning. In addition, it can be said that fixed-partitioning schemes are relatively simple and require minimal OS software and processing overhead. However, there are disadvantages:

- The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system.
- Because partition sizes are preset at system generation time, small jobs will not utilize partition space efficiently. In an environment where the main storage

appears at the end of the memory space, is quickly broken up into small fragments.

The buddy system is a reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes, but in contemporary operating systems, virtual memory based on paging and segmentation is superior. However, the buddy system has found application in parallel systems as an efficient means of allocation and release for parallel programs (e.g., see [JOHN92]). A modified form of the buddy system is used for UNIX kernel memory allocation (described in Chapter 8).

Relocation

Before we consider ways of dealing with the shortcomings of partitioning, we must clear up one loose end, which relates to the placement of processes in memory. When the fixed partition scheme of Figure 7.3a is used, we can expect that a process will always be assigned to the same partition. That is, whichever partition is selected when a new process is loaded will always be used to swap that process back

of the program; these values must be set when the program is loaded into memory or

address (page number, offset), the processor uses the page table to produce a physical address (frame number, offset).

Continuing our example, the five pages of process D are loaded into frames 4, 5, 6, 11, and 12. Figure 7.10 shows the various page tables at this time. A page table contains one entry for each page of the process, so that the table is easily indexed by the page number (starting at page 0). Each page table entry contains the number of the frame in main memory, if any, that holds the corresponding page. In addition, the OS maintains a single free-frame list of all the frames in main memory that are currently unoccupied and available for pages.

Thus we see that simple paging, as described here, is similar to fixed partitioning. The differences are that, with paging, the partitions are rather small; a

the linker. Each logical address (page number, offset) of a program is identical to its relative address. Second, it is a relatively easy matter to implement a function in hardware to perform dynamic address translation at run time. Consider an address of $n + m$ bits, where the leftmost n bits are the page number and the rightmost m bits are the offset. In our example (Figure 7.11b), $n = 6$ and $m = 10$. The following

Let's assume this will be the string START

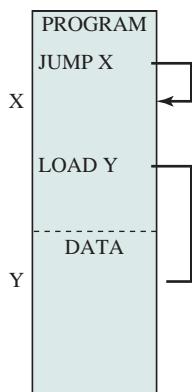
used with C strings.⁶

vulnerable programs. What the attacker does with the resulting corruption of memory varies considerably, depending on what values are being overwritten.

- 7.2** Consider a fixed partitioning scheme with equal-size partitions of 2^{16} bytes and a total main memory size of 2^{24} bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?
- 7.3** Consider a dynamic ired for the pointen42iy bg13h.9(ined,2iy bg1rtiaveragpointen,2iy bg1 form

- 7.12** Consider a simple paging system with the following parameters: 2^{32} bytes of physical memory; page size of 2

memory addresses. For example, if x in Figure 7.16 is location 1024, then the first



(a) Object module

all other memory references within the module are expressed relative to the beginning of the module.

With all memory references expressed in relative format, it becomes a simple task for the loader to place the module in the desired location. If the module is to be loaded beginning at memory address 66.6(), the loader adds 66.6 to each relative address to get the absolute address.

CHAPTER

8.1 A M M ↗

8.1

You're gonna need a bigger boat.

—Steven Spielberg, *JAWS*, 1975

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Define virtual memory.
- Describe the hardware and control structures that support virtual memory.
- Describe the various OS mechanisms used to implement virtual memory.
- Describe the virtual memory management mechanisms in UNIX, Linux, and Windows 7.

Chapter 7 introduced the concepts of paging and segmentation and analyzed their

toprichis oimpicatzedbyd thefacts that memory managementthisas oimpex7 inerrRel-
tib

tib

1.

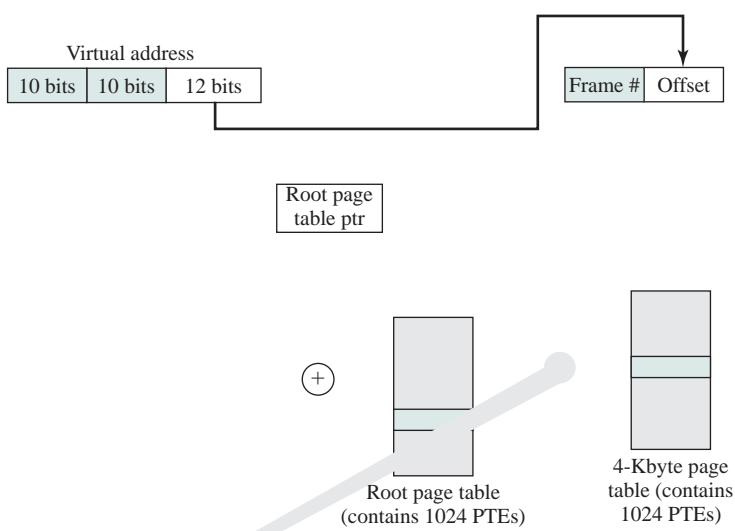
Locality and Virtual Memory

The benefits of virtual memory are attractive, but is the scheme practical? At one

table for a process is created and loaded into main memory. Each page table entry (PTE) contains the frame number of the corresponding page in main memory. A page table is also needed for a virtual memory scheme based on paging. Again, it is typical to associate a unique page table with each process. In this case, however, the page table entries become more complex (Figure 8.2a). Because only some of the pages of a process may be in main memory, a bit is needed in each page table entry to indicate whether the corresponding page is present (P) in main memory or not. If the bit indicates that the page is in memory, then the entry also includes the frame number of that page.

The page table entry includes a modify (M) bit, indicating whether the contents of the corresponding page have been altered since the page was last loaded into main memory. If there has been no change, then it is not necessary to write the page out when it comes time to replace the page in the frame that it currently occupies. Other control bits may also be present. For example, if protection or sharing is managed at the page level, then bits for that purpose will be required.

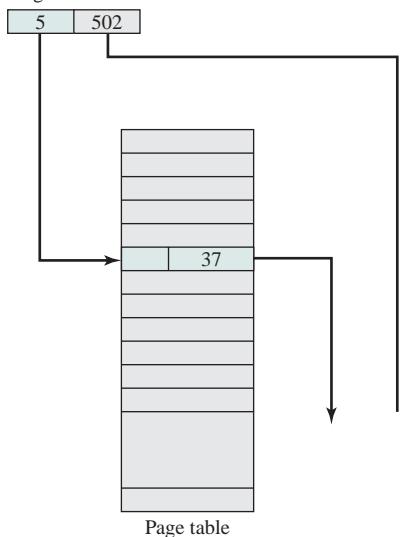
PAGE



translation for this scheme. The root page always remains in main memory. The

Virtual address

Page # Offset



(a) Direct mapping

(b) Associative mapping

be loaded into main memory and its block loaded into the cache. In addition, the

size, the fault rate drops as the number of pages maintained in main memory grows.³

case, however, the segment table entries become more complex (Figure 8.2b). Because only some of the segments of a process may be in main memory, a bit is needed in each segment table entry to indicate whether the corresponding segment is present in main memory or not. If the bit indicates that the segment is in memory, then the entry also includes the starting address and length of that segment.

Another control bit in the segmentation table entry is a modify bit, indicating whether the contents of the corresponding segment have been altered since the segment was last loaded into main memory. If there has been no change, then it is not necessary to write the segment out when it comes time to replace the segment in the frame that it currently occupies. Other control bits may also be present. For example, if protection or sharing is managed at the segment level, then bits for that purpose will be required.

The basic mechanism for reading a word from memory involves the translation

case, lress, consusa wordent table ent. use onlysegment table entri the vg i entrth of ,
mplpon replhold n the regT*ertheIn*ead,

ment
casei

ca(esponding

a base field, which now refers to a page table. The present and modified bits are not needed because these matters are handled at the page level. Other control bits may be used, for purposes of sharing and protection. The page table entry is essentially the same as is used in a pure paging system. Each page number is mapped into a corresponding frame number if the page is present in main memory. The modified bit indicates whether this page needs to be written back out when the frame is allocated to another page. There may be other control bits dealing with protection or other aspects of memory management.

Protection and Sharing

time of its last reference; this would have to be done at each memory reference, both instruction and data. Even if the hardware would support such a scheme, the overhead would be tremendous. Alternatively, one could maintain a stack of page references, again an expensive prospect.

Figure 8.15 shows an example of the behavior of LRU, using the same page address stream as for the optimal policy example. In this example, there are four page faults.

The **first-in-first-out (FIFO)** policy treats the page frames allocated to a proc-

REPLACEMENT SCOPE The scope of a replacement strategy can be categorized as global or local. Both types of policies are activated by a page fault when there are no free page frames. A **local replacement policy** chooses only among the resident pages of the process that generated the page fault in selecting a page to replace. A **global replacement policy**

VARIABLE ALLOCATION, GLOBAL SCOPE This combination is perhaps the easiest to implement and has been adopted in a number of operating systems. At any given

$$\tilde{I}^{-1}(A) \subset A \subset \tilde{I}(A) \subset I(A)$$

sizes alternate with periods of rapid change. When a process first begins executing, it gradually builds up to a working set as it references new pages. Eventually, by the principle of locality, the process should stabilize on a certain set of pages. Subsequent transient periods reflect a shift of the program to a new locality. During

will have their use bit set; these pages are retained in the resident set of the process throughout the next interval, while the others are discarded. Thus the resident set size can only decrease at the end of an interval. During each interval, any faulted

$$= \gamma_{\alpha} \gamma_{\beta} \gamma_{\gamma} A_{\alpha\beta\gamma} + A_{\alpha\beta\gamma} A_{\alpha\beta\gamma} + A_{\alpha\beta\gamma} A_{\alpha\beta\gamma}$$

Table 8.6 UNIX SVR4 Memory Management Parameters

In buddy systems, the cost to allocate and free a block of memory is low compared to that of best-fit or first-fit policies [KNUT97]. However, in the case of kernel memory management, the allocation and free operations must be made as fast as possible. The drawback of the buddy system is the time required to fragment

8.4 LINUX MEMORY MANAGEMENT

Linux shares many of the characteristics of the memory management schemes of other UNIX implementations but has its own unique features. Overall, the Linux memory management scheme is quite complex [DUBE98]. In this section, we give a brief overview of the two main aspects of Linux memory management: process virtual memory and kernel memory allocation.

Linux Virtual Memory

VIRTUAL MEMORY

are allocated and deallocated in main memory, the available groups are split and merged using the buddy algorithm.

PAGE REPLACEMENT ALGORITHM The Linux page replacement algorithm is based on the clock algorithm described in Section 8.2 (see Figure 8.16). In the simple clock algorithm, a use bit and a modify bit are associated with each page in main memory. In the Linux scheme, the use bit is replaced with an 8-bit age variable. Each time that a page is accessed, the age variable is incremented. In the background, Linux periodically sweeps through the global page pool and decrements the age variable for each page as it rotates through all the pages in main memory. A page with an age of 0 is an “old” page that has not been referenced in some time and is the best candidate for replacement. The larger

and AMD64 platforms have 4 Kbytes per page and Intel Itanium platforms have 8 Kbytes per page.

Windows Virtual Address Map

On 32-bit platforms, each Windows user process sees a separate 32-bit address space, allowing 4 Gbytes of virtual memory per process. By default, half of this memory is

- **0x7FFF0000 to 0x7FFFFFFF:** A guard page inaccessible to the user. This page makes it easier for the OS to check on out-of-bounds pointer references.
- **0x80000000 to 0xFFFFFFFF:** System address space. This 2-Gbyte process is used for the Windows Executive, Kernel, HAL, and device drivers.
- On 64-bit platforms, 8 Tbytes of user address space is available in Windows 7.

Windows Paging

When a process is created, it can in principle make use of the entire user space of

pages that have not been recently used from the process. This policy makes



8.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

associative mapping demand paging external fragmentation fetch policy frame hash table hashing internal fragmentation locality	page page fault page placement policy page replacement policy page table paging prepaging real memory resident set	resident set management segment segment table segmentation slab allocation thrashing translation lookaside buffer virtual memory working set
--	--	--

Review Questions

- 8.1** What is the difference between simple paging and virtual memory paging?
- 8.2** Explain thrashing.
- 8.3** Why is the principle of locality crucial to the use of virtual memory?
- 8.4** What elements are typically found in a page table entry? Briefly define each element.
- 8.5** What is the purpose of a translation lookaside buffer?
- 8.6** Briefly define the alternative page fetch policies.
- 8.7** What is the difference between resident set management and page replacement policy?

a.

- 8.15** Consider the following sequence of page references (each element in the sequence represents a page number):

1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5

Define the *mean working set size* after the k th reference as k

I take a two hour nap, from one o'clock to four.

—Yogi Berra

LEARNING OBJECTIVES

to the current set of processes. Each time a job terminates, the scheduler may decide to add one or more new jobs. Additionally, if the fraction of time that the processor is idle exceeds a certain threshold, the long-term scheduler may be invoked.

The decision as to which job to admit next can be on a simple first-come-first-served (FCFS) basis, or it can be a tool to manage system performance. The criteria used may include priority, expected execution time, and I/O requirements. For example, if the information is available, the scheduler may attempt to keep a mix of processor-bound and I/O-bound processes.² Also, the decision can depend on which I/O resources are to be requested, in an attempt to balance I/O usage.

For interactive programs in a time-sharing system, a process creation request can

use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

Short-Term Scheduling

In terms of frequency of execution, the long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process and which one to take. The medium-term scheduler is executed somewhat more frequently to make a swapping decision. The short-term scheduler, also known as the dispatcher, executes most frequently and makes the fine-grained decision of which process to execute next.

The short-term scheduler is invoked whenever an event occurs that may lead

response time may require a scheduling algorithm that switches between processes frequently. This increases the overhead of the system, reducing throughput. Thus, the design of a scheduling policy involves compromising among competing requirements; the relative weights given the various requirements will depend on the nature and intended use of the system.

In most interactive operating systems, whether single user or time shared, adequate response time is the critical requirement. Because of the importance of this requirement, and because the definition of adequacy will vary from one application to another, the topic is explored further in Appendix G.

The Use of Priorities

In many systems, each process is assigned a priority and the scheduler will always choose a process of higher priority over one of lower priority. Figure 9.4 illustrates the use of priorities. For clarity, the queueing diagram is simplified, ignoring the existence of multiple blocked queues and of suspended states (compare Figure 3.8a). Instead

Table 9.5 A Comparison of Scheduling Policies

treatment of processor-bound and I/O-bound processes. Generally, an I/O-bound

main ready queue. Performance studies by the authors indicate that this approach is indeed superior to round robin in terms of fairness.

SHORTEST PROCESS NEXT Another approach to reducing the bias in favor of long processes inherent in FCFS is the shortest process next (SPN) policy. This is a nonpreemptive policy in which the process with the shortest expected processing time is selected next. Thus, a short process will jump to the head of the queue past longer jobs.

Figure 9.5 and Table 9.5 show the results for our example. Note that process

Because both α and $(1 - \alpha)$ are less than 1, each successive term in the preceding equation is smaller. For example, for $\alpha = 0.8$, Equation (9.4) becomes

$$S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \dots + (0.2)^n S_1$$

The older the observation, the less it is counted in to the average.

The size of the coefficient as a function of its position in the expansion is shown in Figure 9.8. The larger the value of α , the greater is the weight given to the more recent observations. For $\alpha = 0.8$, virtually all of the weight is given to the four most recent observations, whereas for $\alpha = 0.2$, the averaging is effectively spread out over the eight or so most recent observations. The advantage of using a value of α close to the eight

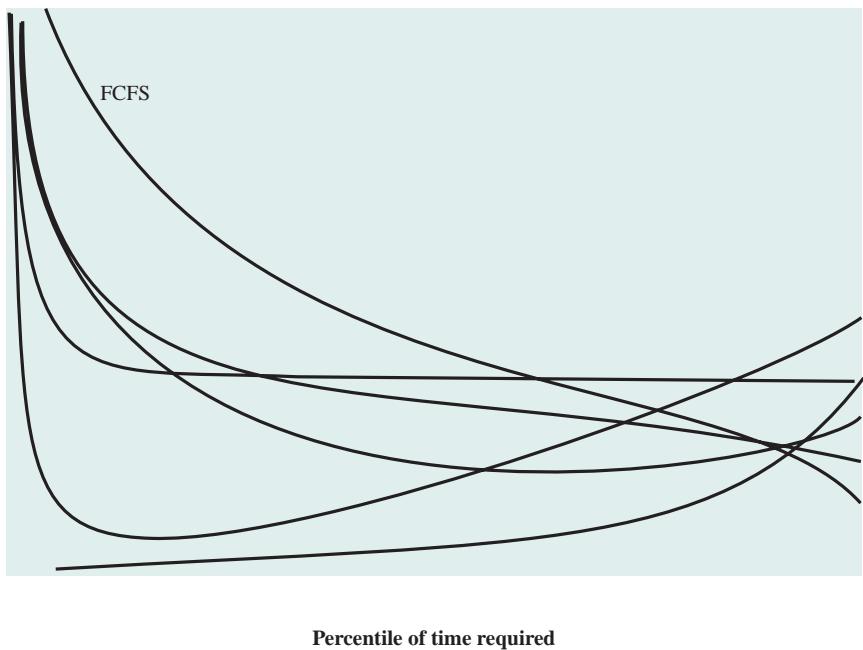
Even with the allowance for greater time allocation at lower priority, a longer process may still suffer starvation. A possible remedy is to promote a process to a higher-priority queue after it spends a certain amount of time waiting for service in its current queue.

Performance Comparison

416

A9 cs1 scn0 Tc0 Tw.0594

Normalized turnaround time



a normalized turnaround time greater than 10 times the service time; furthermore, these are the shortest processes. On the other hand, the absolute waiting time is

group of processes, along with the individual execution history of each process in making scheduling decisions. The system divides the user community into a set of fair-share groups and allocates a fraction of the processor resource to each group. Thus, there might be four groups, each with 25% of the processor usage. In effect, each fair-share group is provided with a virtual system that runs proportionally slower than a full system.

Scheduling is done on the basis of priority, which takes into account the underlying priority of the process, its recent processor usage, and the recent processor usage of the group to which the process belongs. The higher the numerical value of the priority, the lower is the priority. The following formulas apply for process j in group k :

$$CPU_j(i) = \frac{CPU_j(i - 1)}{1 + \alpha \cdot CPU_{j,k}}$$

9.3 TRADITIONAL UNIX SCHEDULING

practical time-sharing scheduling algorithms. The scheduling scheme for SVR4 includes an accommodation for real-time requirements, and so its discussion is deferred to Chapter 10.

The traditional UNIX scheduler employs multilevel feedback using round robin within each of the priority queues. The system makes use of one-second

9.4 SUMMARY

Example for FCFS (1 unit = 10 ms):

P1	P1	P1	P1	P1	P2	P2	P3	P4	P4	P4	P4								
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- 9.10** A processor is multiplexed at infinite speed among all processes present in a ready queue with no overhead. (This is an idealized model of round-robin scheduling)

$$z_j = \sqrt{M} z_{j,M} e^{i\theta_j} \quad (j=1, \dots, M)$$

438

$A_1 \cup A_2 \cup \dots \cup A_n = A$ if and only if $A_i \subseteq A$ for all i .

•

performance gets, because there is a greater frequency of thread preemption and rescheduling. This excessive preemption results in inefficiency from many sources, including time spent waiting for a suspended thread to leave a critical section, time wasted in process switching, and inefficient cache behavior.

The authors conclude that an effective strategy is to limit the number of active threads to the number of processors in the system. If most of the applications are either single thread or can use the task-queue structure, this will provide an effective and reasonably efficient use of the processor resources.

Both dedicated processor assignment and gang scheduling attack the scheduling problem by addressing the issue of processor allocation. One can observe that the processor allocation problem on a multiprocessor more closely resembles the memory allocation problem on a uniprocessor than the scheduling problem on a uniprocessor. The issue is how many processors to assign to a program at any given time, which is analogous to how many page frames to assign to a given process at any time. [GEHR87] proposes the term

DYNAMIC SCHEDULING For some applications, it is possible to provide language and system tools that permit the number of threads in the process to be altered

possible. For example, a typical traditional UNIX system, when it detects a corrup-

450

$$A_{\text{left}} \cup A_{\text{right}} = A_{\text{left}} \cap A_{\text{right}} = \emptyset$$



earliest deadline, it is scheduled first. When A1 completes, B1 is given the processor.
At $t =$

Figure 10.8 illustrates the relevant parameters for periodic tasks. The task's period, T , is the amount of time between the arrival of one instance of the task and the arrival of the next instance of the task. A task's rate (in hertz) is simply the inverse of its period (in seconds). For example, a task with a period of 50 ms occurs at a rate of 20 Hz. Typically, the end of a task's period is also the task's hard deadline, although some tasks may have earlier deadlines. The execution (or computation) time, C

454

A task with utilization of 0.2, A task with utilization of 0.267, and a task with utilization of 0.286.

The total utilization of these three tasks is $0.2 + 0.267 + 0.286 =$

resource; it should end when the resource is released by the lower-priority task. Figure 10.9b shows that priority inheritance resolves the problem of unbounded priority inversion illustrated in Figure 10.9a. The relevant sequence of events is as follows:

$t_1:T_3$ begins executing.

$t_2:T_3$

For FIFO threads, the following rules apply:

1. The system will not interrupt an executing FIFO thread except in the following cases:
 - a. Another FIFO thread of higher priority becomes ready.
 - b. The executing FIFO thread becomes blocked waiting for an event, such as I/O.
 - c. The executing FIFO thread voluntarily gives up the processor following a call to the primitive `sched_yield`.
2. When an executing FIFO thread is interrupted, it is placed in the queue associated with its priority.
3. When a FIFO thread becomes ready and if that thread has a higher priority than the currently executing thread, then the currently executing thread is preempted and the highest-priority ready FIFO thread is executed. If more than one thread

Non-Real-Time Scheduling

The Linux 2.4 scheduler for the SCHED_OTHER

Priority class	Global
Real time	
Kernel	
Time shared	

processes can make use of preemption points to preempt kernel processes and user processes.

- **Kernel (99-60):** Processes at these priority levels are guaranteed to be selected to run before any time-sharing process but must defer to real-time processes.
 - **Time-shared (59-0):** The lowest-priority processes, intended for user applications other than real-time applications.

Figure 10.13 indicates how scheduling is implemented in SVR4. A dispatch queue is associated with each priority level, and processes at a given priority level are executed in round-robin fashion. A bit-map vector, dqactmap, contains one bit for each priority level; the bit is set to one for any priority level with a nonempty queue. Whenever a running process leaves the Running state, due to a block, timeslice expiration, or preemption, the dispatcher checks dqactmap and dispatches a ready process from the highest-priority nonempty queue. In addition, whenever a defined preemption point is reached, the kernel checks a flag called kprunrun. If set, this indicates that at least one real-time process is in the Ready state, and the kernel preempts the current process if it is of lower priority than the highest-priority real-time ready process.

Within the time-sharing class, the priority of a process is variable. The scheduler reduces the priority of a process each time it uses up a time quantum, and it raises its

it has used up its current time quantum, the kernel lowers its priority. Thus, processor-bound threads tend toward lower priorities and I/O-bound threads tend toward higher priorities. In the case of I/O-bound threads, the kernel boosts the priority more (higherased upthreads on keyboarddisleads) - its (e.g., onward

This equation denotes the fraction of a single processor in a system. Thus, for example, if a system is multicore with four cores and we wish to provide one VM on an average of one dedicated processor, then we set $R = 1$ and $T = 4$. The overall system is limited as follows. If there are N VMs, then:

a

its entire life or dispatched to any processor each time it enters the Running state. Performance studies suggest that the differences among various scheduling algorithms are less significant in a multiprocessor system.

10.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

aperiodic task
deadline scheduling
deterministic operating system
fail-soft operation
gang scheduling
granularity

hard real-time task
load sharing
periodic task
priority inversion
rate monotonic scheduling
real-time operating system

real-time scheduling priority 0 (operation)-

- 10.2** Consider a set of five aperiodic tasks with the execution profiles of Table 10.7. Develop scheduling diagrams similar to those of Figure 10.6 for this set of tasks.
- 10.3** Least laxity first (LLF) is a real-time scheduling algorithm for periodic tasks. Slack time, or laxity, is the amount of time between when a task would complete if it started now and its next deadline. This is the size of the available scheduling window. Laxity can be expressed as

L

- Machine readable:

evident than in the I/O function. The evolutionary steps can be summarized as follows:

1. The processor directly controls a peripheral device. This is seen in simple microprocessor-controlled devices.
2. A controller or I/O module is added. The processor uses programmed I/O without interrupts. With this step, the processor becomes somewhat divorced from the specific details of external device interfaces.
3. The same configuration as step 2 is used, but now interrupts are employed. The processor need not spend time waiting for an I/O operation to be performed, thus increasing efficiency.
4. The I/O module is given direct control of memory via DMA. It can now move a block of data to or from memory without involving the processor, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a separate processor, with a CPU control function. This not only speeds execution but also reduces the load on the CPU.

The DMA technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module
- The address of the I/O device involved, communicated on the data lines
-

system bus. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus

extremely slow compared with main memory and the processor. One way to tackle



portions of the process may be paged out to disk, it is impossible to swap the process out completely, even if this is desired by the operating system. Notice also that there is a risk of single-process deadlock. If a process issues an I/O command, is suspended awaiting the result, and then is swapped out prior to the beginning of the operation, the process is blocked waiting on the I/O event, and the I/O operation is blocked waiting for the process to be swapped in. To avoid this deadlock, the user memory involved in the I/O operation must be locked in main memory immediately

it hardly makes sense to queue disk writes to the same device for swapping the process out. This attempt to swap the process and release main memory will itself not begin until after the I/O operation finishes, at which time swapping the process to disk may no longer be appropriate.

Similar considerations apply to block-oriented output. When data are being transmitted to a device, they are first copied from the user space into the system buffer, from which they will ultimately be written. The requesting process is now

of terminal, user input is one line at a time, with a carriage return signaling the end of a line, and output to the terminal is similarly one line at a time. A line printer is another example of such a device. Byte-at-a-time operation is used on forms-mode terminals, when each keystroke is significant, and for many other peripherals, such as sensors and controllers.

In the case of line-at-a-time I/O, the buffer can be used to hold a single line. The user process is suspended during input, awaiting the arrival of the entire line. For output, the user process can place a line of output in the buffer and continue

service. Even with multiple buffers, all of the buffers will eventually fill up and the process will have to wait after processing each chunk of data. However, in a multi-programming environment, when there is a variety of I/O activity and a variety of process activity to service, buffering is one tool that can increase the efficiency of the OS and the performance of individual processes.

11.5 DISK SCHEDULING

Over the last 40 years, the increase in the speed of processors and main memory has far outstripped that for disk access, with processor and main memory speeds increasing by about two orders of magnitude compared to one order of magnitude for disk. The result is that disks are currently at least four orders of magnitude slower than main memory. This gap is expected to continue into the foreseeable future. Thus, the performance of disk storage subsystem is of vital concern, and much research has gone into schemes for improving that performance. In this section, we highlight some of the key issues and look at the most important approaches. Because the performance of the disk system is tied closely to file system design issues, the discussion continues in Chapter 12.

Disk Performance Parameters

The actual details of disk I/O operation depend upon the computer

Thus, the total average access time can be expressed as

$$T_a = T_s + \frac{1}{n}$$



Figure 11.7b and Table 11.2b show the performance of SSTF on the same example as was used for FIFO. The first track accessed is 90, because this is the closest requested track to the starting position. The next track accessed is 58 because this is the closest of the remaining requested tracks to the current position of 90. Subsequent tracks are selected accordingly.

SCAN With the exception of FIFO, all of the policies described so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request. A exists as wT^*

Table 11.4 RAID Levels

This is best understood by considering Figure 11.8. All user and system data are

Suppose that drive X1 has failed. If we add $X4(i) \oplus X1(i)$ to both sides of the preceding equation, we get

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

Thus, the contents of each strip of data on X1 can be regenerated from the contents of the corresponding strips on the remaining disks in the array. This principle is true for RAID levels 3 through 6.



processor. Such a cache memory reduces average memory access time by exploiting the principle of locality.

The same principle can be applied to disk memory. Specifically, a disk cache is a buffer in main memory for disk sectors. The cache contains a copy of some of the sectors on the disk. When an I/O request is made for a particular sector, a check is made to determine if the sector is in the disk cache. If so, the request is satisfied via the cache. If not, the requested sector is read into the disk cache from the disk. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single I/O request, it is likely that there will be future

To overcome this difficulty with LFU, a technique known as frequency-based replacement is proposed in [ROBI90]. For clarity, let us first consider a simplified version, illustrated in Figure 11.9a. The blocks are logically organized in a stack, as with the LRU algorithm. A certain portion of the top part of the stack is designated

relatively frequently referenced blocks a chance to build up their reference counts

references that map into the same hash table entry, if the corresponding block is in the buffer cache, then that buffer will be in the chain for that hash table entry. Thus, the length of the search of the buffer cache is reduced by a factor on the order of N , where $1 \text{ Tf}34.975 \text{ 0.003 Tc}0 \text{ che} 20 \text{ Tw}(N)\text{Tj}1 \text{ Tf}0.775 \text{ 0 TD-0.25 Tw}[(\text{,0029N})\text{Tj}(\text{,0001}]TJ-35$



elevator queue, as before. In addition, the same request is placed at the tail of a read FIFO queue for a read request or a write FIFO queue for a write request. Thus, the read and write queues maintain a list of requests in the sequence in which the requests were made. Associated with each request is an expiration time, with a default value of 0.5 seconds for a read request and 5 seconds for a write request. Ordinarily, the scheduler dispatches from the sorted queue. When a request is satisfied, it is removed from the head of the sorted queue and also from the appropriate FIFO queue. However, when the item at the head of one of the FIFO queues becomes older than its expiration time, then the scheduler next dispatches from that FIFO queue, taking the expired request, plus the next few requests from the queue. As each request is dispatched, it is also removed from the sorted queue.

The deadline I/O scheduler scheme overcomes the starvation problem and also the read versus write problem.

ANTICIPATORY I/O S

In Linux, the anticipatory scheduler is superimposed on the deadline sched-

MEE96b Mee, C., and Daniel, E. eds. *Magnetic Storage Handbook*. New York: McGraw Hill, 1996.

NG98 Ng, S. "Advances in Disk Technology: Performance Issues."

11.13 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Problems

- 11.1** Consider a program that accesses a single I/O device and compare unbuffered I/O to

1

file design, fields may be fixed length or variable length. In the latter case, the field often consists of two or three subfields: the actual value to be stored, the name of the field, and, in some cases, the length of the field. In other cases of variable-length fields, the length of the field is indicated by the use of special demarcation symbols between fields.

A record

524

A. $\frac{1}{2}x^2 - 1 = \frac{1}{2}(A - A)$

- Delete_One

assigned and secondary memory is allocated at this level. The basic I/O supervisor is part of the operating system.

Logical I/O enables users and applications to access records. Thus, whereas the basic file system deals with blocks of data, the logical I/O module deals with file records. Logical I/O provides a general-purpose record I/O capability and maintains basic data about files.

The level of the file system closest to the user is often termed the **access method**. It provides a standard interface between applications and the file systems

- The pile
- The sequential file
- The indexed sequential file

record is found or the entire file has been searched. If we wish to find all records that contain a particular field or contain that field with a particular value, then the entire file must be searched.

the key field, both forms of sequential file are inadequate. In some applications, the flexibility of efficiently searching by various attributes is desirable.

Before illustrating the concept of B-tree, let us define a B-tree and its characteristics more precisely. A B-tree is a tree structure (no closed loops) with the following characteristics (Figure 12.4).

1. The tree consists of a number of nodes and leaves.
2. Each node contains at least one key which uniquely identifies a file record, and more than one pointer to child nodes or leaves. The number of keys and pointers contained in a node may vary, within limits explained below.
3. Each node is limited to the same number of maximum keys.
4. The keys in a node are stored in nondecreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node. Thus, each node has one more pointer than keys.

Structure

The way in which the information of Table 12.2 is stored differs widely among various systems. Some of the information may be stored in a header record associated with the file; this reduces the amount of storage required for the directory,

these user directories, in turn, may have subdirectories and files as entries. This is true at any level: That is, at any level, a directory may consist of entries for subdirec-

Although the pathname facilitates the selection of file names, it would be awkward for a user to have to spell out the entire pathname every time a reference is made to a file. Typically, an interactive user or a process has associated with it a current directory, often referred to as the **working directory**. Files are then referenced relative to the working directory. For example, if the working directory for user B is "Word," then the pathname `Unit_A/ABC` is sufficient to identify the file in the lower left-hand corner of Figure 12.7. When an interactive user logs on, or

12.5 FILE SHARING

In a multiuser system, there is almost always a requirement for allowing files to be

One user is designated as owner of a given file, usually the person who initially created a file. The owner has all of the access rights listed previously and may grant rights to others. Access can be provided to different classes of users:

- **Specific user:**

access, degree of multiprogramming, other performance factors in the system, disk caching, disk scheduling, and so on.

FILE ALLOCATION METHODS Having looked at the issues of preallocation versus dynamic allocation and portion size, we are in a position to consider specific file allocation methods. Three methods are in common use: contiguous, chained, and

BIT TABLES This method uses a vector containing one bit for each block on the disk. Each entry of a 0 corresponds to a free block, and each 1 corresponds to a block in use. For example, for the disk layout of Figure 12.9, a vector of length 35 is needed and would have the following value:

0011100001111100001111111111011000

A bit table has the advantage that it is relatively easy to find one or a con-

However, it can still be sizable. The amount of memory (in bytes) required for a block bitmap is

$$\frac{\text{disk size in bytes}}{8}$$

the disk. Depending on the size of the disk, either 24 or 32 bits will be needed to

however, must control access to specific records or even portions of records. For example, it may be permissible for anyone in administration to obtain a list of company personnel, but only selected individuals may have access to salary information.

[REDACTED]

- The size of the extended attribute information
- Zero or more extended attribute entries

The blocksize value is typically the same as, but sometimes larger than, the file system blocksize. On traditional UNIX systems, a fixed blocksize of 512 bytes was used. FreeBSD has a minimum blocksize of 4,096 bytes (4 Kbytes); the blocksize can be any power of 2 greater than or equal to 4,096. For typical file systems, the blocksize is 8 Kbytes or 16 Kbytes. The default FreeBSD blocksize is 16 Kbytes.

Extended attribute entries are variable-length entries used to store auxiliary data that are separate from the contents of the file. The first two extended attributes defined for FreeBSD deal with security. The first of these support access control

2. Smaller files may be accessed with little or no indirection, reducing processing and disk access time.

The remaining three bits define special additional behavior for files or direc-

mapping module is needed to transform the characteristics of the real file system to the characteristics expected by the virtual file system.

Figure 12.19 indicates the key ingredients of the Linux file system strategy. A user process issues a file system call (e.g., read) using the VFS file scheme. The VFS converts this into an internal (to the kernel) file system call that is passed to a mapping function for a specific file system [e.g., IBM's Journaling File System (JFS)]. In most cases, the mapping function is simply a mapping of file system functional calls from one scheme to another. In some cases, the mapping function is more complex. For example, some file systems use a file allocation table (FAT), which stores the

system. The VFS is independent of any file system, so the implementation of a mapping function must be part of the implementation of a file system on Linux. The target file system converts the file system request into device-oriented instructions that are passed to a device driver by means of page cache functions.

VFS is an object-oriented scheme. Because it is written in C, rather than a

The Dentry Object

for changes to the file system; each significant change is treated as an atomic action that is either entirely performed or not performed at all. Each transac-

software RAID 5 is employed, a volume consists of stripes spanning multiple disks. The maximum volume size for NTFS is 264 bytes.

The cluster is the fundamental unit of allocation in NTFS, which does not recognize sectors. For example, suppose each sector is 512 bytes and the system is configured with two sectors per cluster (one cluster =

that alters important file system data structures is recorded in a log file before being recorded on the disk volume. Using the log, a partially completed transaction at the time of a crash can later be redone or undone when the system recovers.

In general terms, these are the steps taken to ensure recoverability, as described in [RUSS11]:

1. NTFS first calls the log file system to record in the log file (in the cache) any transactions that will modify the volume structure.
2. NTFS modifies the volume (in the cache).
3. The cache manager calls the log file system to prompt it to flush the log file to disk.
- 4.

12.13 RECOMMENDED READING

There are a number of good books on file structures and file management. The following all focus on file management systems but also address related OS issues. Perhaps the most useful is [WIED87], which takes a quantitative approach to file management and deals with all of the issues raised in Figure 12.2, from disk scheduling to file structure. [VENU09] presents an object-oriented design approach toward file structure implementation. [LIVA90] emphasizes file structures, providing a good and lengthy survey with comparative performance analyses. [GROS86] pro-

12.14 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

- 12.3** What file organization would you choose to maximize efficiency in terms of speed of access, use of storage space, and ease of updating (adding/deleting/modifying) when

In brief, the conventional arguments that bird brains are too small or do not have particular structures needed for intelligence are based on ignorance of brains in general and bird brains in particular. It is unwarranted to argue that the small brains and small bodies of birds render them less capable of behaving with intelligent awareness than animals with large brains and large bodies.

T-

- Relaxed to very strict and difficult requirements, (with 1842 in) Epal tests safety of individual quality

The following are some of the unique characteristics and design requirements for embedded operating systems:

-

- Provides for special alarms and time-outs
- Supports real-time queuing disciplines such as earliest deadline first and

Figure 13.2 shows the top level of the eCos configuration tool as seen by the tool user. Each of the items on the list in the left-hand window can be

Figure 13.3 eCos Config

Note that the HAL interface can be directly used by any of the upper layers, promoting efficient code.

eCos Kernel The eCos kernel was designed to satisfy four main objectives:

-

4. A range of synchronization primitives, allowing threads to interact and share data safely
5. Integration with the system's support for interrupts and exceptions

Some functionality that is typically included in the kernel of an OS is not

eCos Scheduler

The eCos kernel can be configured to provide one of two scheduler designs: the bitmap scheduler and a multilevel queue scheduler. The configuration user selects the appropriate scheduler for the environment and the application. The bitmap scheduler provides efficient scheduling for a system with a small number of threads.




```
cyg_mutex_t res_lock;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;
void res_init(void)
{
    cyg_mutex_init(&res_lock);
    <fill pool with resources>
}
res_t res_allocate(void)
{
    res_t res;

    cyg_mutex_lock(&res_lock);                                // lock the mutex
    if( res_count == 0 )                                      // check for free resource
```

Figure 13.8 Controlling Access to a Pool of Resources Using Mutexes

Section 5.5 provides a general discussion of message-passing synchronization. Here, we look at the specifics of the eCos version.

The eCos mailbox mechanism can be configured for blocking or nonblocking on both the send and receive side. The maximum size of the message queue associated with a given mailbox can also be configured.

The send message primitive, called `put`, includes two arguments: a handle to the mailbox and a pointer for the message itself. There are three variants to this primitive:

`cyg_mbox_put`

If there is a spare slot in the mailbox, then the new message is placed there; if there is a waiting thread, it will be woken up so that it can receive the message. If the mailbox is currently full, `cyg_mbox_put` blocks until there has been a corresponding get operation and a slot ids available.

`cyg_mbox_timed_put`

system with preemptive scheduling, in which a higher-priority thread attempts to



station connects the sensor network to a host PC and passes on sensor data from the network to the host PC, which can do data analysis and/or transmit the data over a corporate network or Internet to an analysis server. Individual sensors collect data and transmit these to the base station, either directly or through sensors that act as data relays. Routing functionality is needed to determine how to relay the data through the sensor network to the base station. [BUON01] points out that, in many applications, the user will want to be able to quickly deploy a large number of low-cost devices without having to configure or manage them. This means that they must be capable of assembling themselves into an ad hoc network. The mobility of individual sensors and the presence of RF interference means that the network will have to be capable of reconfiguring itself in a matter of seconds.

TinyOS Goals

With the tiny, distributed sensor application in mind, a group of researchers from UC Berkeley [HILL00] set the following goals for TinyOS:

- **Allow high concurrency:** In a typical wireless sensor network application, the devices are concurrency intensive. Several different flows of data must be kept moving simultaneously. While sensor data are input in a steady stream, processed results must be transmitted in a steady stream. In addition, external controls from remote sensors or base stations must be managed.
- **Operate with limited resources:** The target platform for TinyOS will have limited memory and computational resources and run on batteries or solar power.

A single platform may offer only kilobytes of program memory and hundreds

special-purpose components and link and load all of the components needed for the user's application. TinyOS, then, consists of a suite of standardized components. Some but not all of these components are used, together with application-specific user-written components, for any given implementation. The OS for that implementation is simply the set of standardized components from the TinyOS suite.

All components in a TinyOS configuration have the same structure, an example of which is shown in Figure 13.11a. The shaded box in the diagram indicates the component, which is treated as an object that can only be accessed by defined interfaces, indicated by white boxes. A component may be hardware or software. Software components are implemented in nesC, which is an extension of C with two distinguishing features: a programming model where components interact via interfaces, and an event-based concurrency model with run-to-completion task and interrupt handlers, explained subsequently.

The architecture consists of a layered arrangement of components. Each com-


```
interface Timer {  
    command result_t start(char type, uint32_t interval);  
    command result_t stop();  
    event result_t fired();  
}  
interface Clock {  
    command result_t setRate(char interval, char scale);  
    event result_t fire();  
}
```

Components are organized into configurations by “wiring” them together at their interfaces and equating the interfaces of the configuration with some of the interfaces of the components. A simple example is shown in Figure 13.11b. The uppercase C stands for Component. It is used to distinguish between an interface (e.g., Timer) and a component that provides the interface (e.g., TimerC). The uppercase M stands for Module. This naming convention is used when a single logicalinterval);oth()];TJT*-0.0234 T

600


```
1 unsigned char buffer_empty = true;
2 cyg_mutex_t mut_cond_var;
3 cyg_cond_t cond_var;
4
```

Figure 13.14 Condition Variable Example Code

This definition introduces three key objectives that are at the heart of computer security:

- **Confidentiality:** This term covers two related concepts:
 - **Data¹ confidentiality:** Assures that private or confidential information is not made available or disclosed to unauthorized individuals

of these three objectives in terms of requirements and the definition of a loss of security in each category:

- **Confidentiality:** Preserving authorized restrictions on infos-18(.2e8(cess)22.5)IT

- **Intrusion:** An example of intrusion is an adversary gaining unauthorized access to sensitive data by overcoming the system's access control protections.

Deception is a threat to either system integrity or data integrity. The following types of attacks can result in this threat consequence:

- **Masquerade:** One example of masquerade is an attempt by an unauthorized user to gain access to a system by posing as an authorized user; this could happen if the unauthorized user has learned another user's logon ID and password. Another example is malicious logic, such as a Trojan horse,

Threats and Assets

The assets of a computer system can be categorized as hardware, software, data, and communication lines and networks. In this subsection, we briefly describe these four categories and relate these to the concepts of integrity, confidentiality, and availability introduced in Section 14.1 (see Figure 14.2 and Table 14.2).

HARDWARE A major threat to computer system hardware is the threat to availability. Hardware is the most vulnerable to attack and the least susceptible to automated controls. Threats include accidental and deliberate damage to equipment as well as theft. The proliferation of personal computers and workstations and the widespread use of LANs increase the potential for losses in potential damage.

break-in at a large financial institution reported in [RADC04]. The intruder took

One of the results of the growing awareness of the intruder problem has been the establishment of a number of computer emergency response teams (CERTs). These cooperative ventures collect information about system vulnerabilities and disseminate it to systems managers. Hackers also routinely read CERT reports. Thus, it is important for system administrators to quickly insert all software patches to discovered vulnerabilities. Unfortunately, given the complexity of many IT systems and the rate at which patches are released, this is increasingly difficult to achieve without automated updating. Even then, there are problems caused by incompatibilities resulting from the updated software (hence the need for multiple layers of defense in managing security threats to IT systems).

CRIMINALS Organized groups of hackers have become a widespread and common threat to Internet-based systems. These groups can be in the employ of a corporation or government but often are loosely affiliated gangs of hackers. Typically, these gangs are young, often Eastern European, Russian, or southeast Asian hackers who

portrayed in the movie *War Games*. Another example is that during the development of Multics, penetration tests were conducted by an Air Force “tiger team” (simulating adversaries). One tactic employed was to send a bogus operating system

computer's memory [TIME90]. The Trojan horse was implanted in a graphics routine offered on an electronic bulletin board system.

Trojan horses fit into one of three models:

-

$$-\beta\tau^2\left(\rho_{\rm{eff}}\right)^2\left(1+\sqrt{\frac{M}{A}}\right)\leq 1\,.$$

During its lifetime, a typical virus goes through the following four phases:

- **Dormant phase:** The virus is idle. The virus will eventually be activated by some event, such as a date, the presence of another program or file, or the capacity of the disk exceeding some limit. Not all viruses have this stage.
- **Propagation phase:** The virus places an identical copy of itself into other programs or into certain system areas on the computer. It may also spread to other disks or networks.
- **Manifestation phase:** The virus becomes active and performs its destructive or annoying function.
- **Termination phase:** The virus ends its activity and disappears from the system.

3. The compressed version of the original infected program, P'_1 , is uncompressed.
4. The uncompressed original program is executed.

In this example, the virus does nothing other than propagate. As previously mentioned, the virus may include a logic bomb.

INITIAL INFECTION Once a virus has gained entry to a system by infecting a single program, it is in a position to potentially infect some or all other executable files on that system when the infected program executes. Thus, viral infection can be completely prevented by preventing the virus from gaining entry in the first place.

- **Stealth virus:** A form of virus explicitly designed to hide itself from detection

To replicate itself, a network worm uses some sort of network vehicle. Examples include the following:

- **Electronic mail facility:** A worm mails a copy of itself to other systems, so that its code is run when the e-mail or an attachment is received or viewed.
- **Remote execution capability:** A worm executes a copy of itself on another

STATE OF WORM TECHNOLOGY The state of the art in worm technology includes the following:

- **Multiplatform:** Newer worms are not limited to Windows machines but can attack a variety of platforms, especially the popular varieties of UNIX.
- **Multiexploit:** New worms penetrate systems in a variety of ways, using exploits against Web servers, browsers, e-mail, file sharing, and other network-based applications.
- **Ultrafast spreading:** One technique to accelerate the spread of a worm is to

Bots

A bot (robot), also known as a zombie or drone, is a program that secretly takes

•

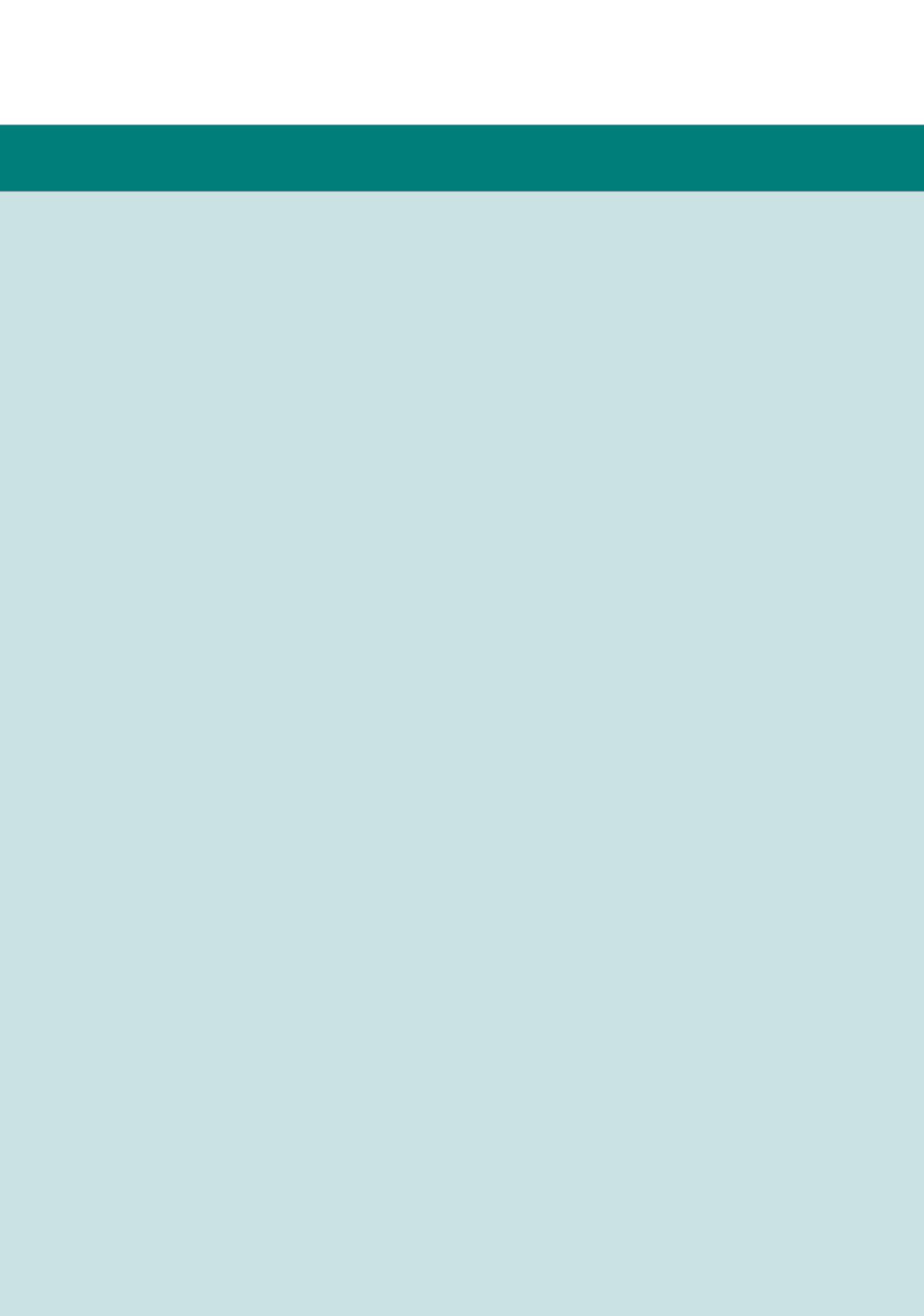
Recommended Web sites:

Review Questions

- 14.1** Define *computer security*.
14.2

- 14.4** For each of the following assets, assign a low, moderate, or high impact level for the

- b.** Answer the same questions for the following program:



•

password of up to eight printable characters in length. This is converted into a 56-bit

Memory cards can be used alone for physical access, such as a hotel room. For computer user authentication, such cards are typically used with some form of pass-

For user authentication to computer, the most important category of smart token is the smart card, which has the appearance of a credit card, has an electronic interface, and may use any of the type of protocols just described. The remainder of this section discusses smart cards.

A smart card contains within it an entire microprocessor, including processor, memory, and I/O ports. Some versions incorporate a special coprocessing circuit for

current contents of the matrix. In addition, certain subjects have the authority to make specific changes to the access matrix. A request to modify the access matrix is treated as an access to the matrix, with the individual entries in the matrix treated as objects. Such accesses are mediated by an access matrix controller, which controls updates to the matrix.

The model also includes a set of rules that govern modifications to the access matrix, shown in Table 15.1. For this purpose, we introduce the access rights *owner* and *control* and the concept of a copy flag, explained in the subsequent paragraphs.

The first three rules deal with transferring, granting, and deleting access rights. Suppose that the entry 0/F50 0 10 90 2096766 496 Tm0.0009241[()26,1666ucea2 2/1

or without copy flag, to another subject. Rule R1 expresses this capability. A subject would transfer the access right without the copy flag if there were a concern that the

environments frequently, and the assignment of a user to one or more roles may also be dynamic. The set of roles in the system in most environments is likely to be

In Anderson's study [ANDE80], it was postulated that one could, with reasonable confidence, distinguish between a masquerader and a legitimate user. Patterns

3. Because of the simple, uniform structure of the detection-specific audit records, it may be relatively easy to obtain this information or at least part of it by a straightforward mapping from existing native audit records to the detection-specific audit records.

At the start of each simulation, the emulator begins interpreting instructions in the target code, one at a time. Thus, if the code includes a decryption routine that decrypts and hence exposes the virus, that code is interpreted. In effect, the virus does the work for the antivirus program by exposing the virus. Periodically, the control module interrupts interpretation to scan the target code for virus signatures.

During interpretation, the target code can cause no damage to the actual personal computer environment, because it is being interpreted in a completely controlled environment.

The most difficult design issue with a GD scanner is to determine how long to run each interpretation. Typically, virus elements are activated soon after a program begins executing, but this need not be the case. The longer the scanner emulates a particular program, the more likely it is to catch any hidden viruses. However, the antivirus program can take up only a limited amount of time and resources before

2. The administrative machine encrypts the sample and sends it to a central virus analysis machine.
3. This machine creates an environment in which the infected program can be safely run for analysis. Techniques used for this purpose include emulation, or the creation of a protected environment within which the suspect program can be executed and monitored. The virus analysis machine then produces a prescription for identifying and removing the virus.
4. The resulting prescription is sent back to the administrative machine.
5. The administrative machine forwards the prescription to the infected client.
6. The prescription is also forwarded to other clients in the organization.
7. Subscribers around the world receive regular antivirus updates that protect them from the new virus.

The success of the digital immune system depends on the ability of the virus analysis machine to detect new and innovative virus strains. By constantly analyzing and monitoring the viruses found in the wild, it should be possible to continually update the digital immune software to keep up with the threat.

BEHAVIOR-BLOCKING SOFTWAREU0F4 1 99.7 Unlik be-30810.heurinii9 ce-30810. foe-30810.f

- Modifications to the logic of executable files or macros;
- Modification of critical system settings, such as start-up settings;
- Scripting of e-mail and instant messaging clients to send executable content; and
- Initiation of network communications.

Figure 15.10 illustrates the operation of a behavior blocker. Behavior-blocking software runs on server and desktop computers and is instructed through policies set by the network administrator to let benign actions take place but to intercede when unauthorized or suspicious actions occur. The module blocks any suspicious software from executing. A blocker isolates the code in a sandbox, which restricts the code's access to various OS resources and applications. The blocker then sends an alert.

Because a behavior blocker can block suspicious software in real time, it has an advantage over such established antivirus detection techniques as fingerprinting or heuristics. While there are literally trillions of different ways to obfuscate and rearrange the instructions of a virus or worm, many of which will evade detection by a fingerprint scanner or heuristic, eventually malicious code must make a well-defined request to the operating system. Given that the behavior blocker can intercept all such requests, it can identify and block malicious actions regardless of how obfuscated the program logic appears to be.

before it has been detected and blocked. For example, a new virus might shuffle a number of seemingly unimportant files around the hard drive before infecting a single

C. Payload-classification-based worm containment: These network-based

value be unpredictable and should be different on different systems. If this were not the case, the attacker would simply ensure the shellcode included the correct canary value in the required location. Typically, a random value is chosen as the canary value on process creation and saved as part of the processes state. The code added to the function entry and exit then uses this value.

There are some issues with using this approach. First, it requires that all programs needing protection be recompiled. Second, because the structure of the stack

- **Group SIDs:** A list of the groups to which this user belongs. A group is simply

The remaining two bits in the access mask have special meanings. The Access_

Problems

- 15.9** For the DAC model discussed in Section 15.2, an alternative representation of the protection state is a directed graph. Each subject and each object in the protection state is represented by a node (a single node is used for an entity that is both subject

- **Client-based processing:** At the other extreme, virtually all application processing may be done at the client, with the exception of data validation routines and other database logic functions that are best performed at the server. Generally, some of the more sophisticated database logic functions are housed on the client side. This architecture is perhaps the most common client/server approach in current use. It enables the user to employ applications tailored to local needs.
- **Cooperative processing:**

benefit of the fat client model is that it takes advantage of desktop power, offloading application processing from servers and making them more efficient and less likely to be bottlenecks.

There are, however, several disadvantages to the fat client strategy. The addition of more functions rapidly overloads the capacity of desktop machines, forcing companies to upgrade. If the model extends beyond the department to incorporate many users, the company must install high-capacity LANs to support the large

any changes to a file back to the server, then any other client that has a cache copy of the relevant portion of the file will have obsolete data. The problem is made even worse if the client delays writing back changes to the server. In that case, the server itself has an obsolete version of the file, selr thnewile wireadirequestto the server.

MIDDLEWARE ARCHITECTURE Figure 16.8 suggests the role of middleware in a client/server architecture. The exact role of the middleware component will depend on the style of client/server computing being used. Referring back to Figure 16.5, recall that there are a number of different client/server approaches, depending on the way in which application functions are split up. In any case, Figure 16.8 gives a good general idea of the architecture involved.

Note that there is both a client and server component of middleware. The

over HTTP (Hypertext Transfer Protocol), known as *Web services*. SOAs are also implemented using other standards, such as CORBA (Common Object Request Broker Architecture).

At a top level, an SOA contains three types of architectural elements [BIH06], illustrated in Figure 16.10:

- **Service provider:**



$$-\gamma^2\left(\partial_{\mu}\partial_{\nu}A_{\alpha}-\partial_{\nu}\partial_{\mu}A_{\alpha}\right)+\partial_{\mu}\partial_{\nu}A_{\alpha}\partial_{\mu}\partial_{\nu}A_{\alpha}$$

require confirmation that a message has been delivered, the applications themselves may use request and reply messages to satisfy the requirement.

Blocking versus Nonblocking

With nonblocking, or asynchronous, primitives, a process is not suspended as a result of issuing a Send or Receive. Thus, when a process issues a Send primi-

Synchronous versus Asynchronous

A clearer picture of the range of clustering approaches can be gained by looking at functional alternatives. A white paper from Hewlett Packard [HP96] provides a useful classification along functional lines (Table 16.2), which we now discuss.

A common, older method, known as

In one approach to clustering, each computer is a

each time with a different set of starting conditions or parameters. A good example is a simulation model, which will run a large number of different scenarios and then develop statistical summaries of the results. For this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an orderly manner.

Cluster Computer Architecture

Figure 16.16 shows a typical cluster architecture. The individual computers are connected by some high-speed LAN or switch hardware. Each computer is capable of operating independently. In addition, a middleware layer of software is installed in each computer to enable cluster operation. The cluster middleware provides a unified system image to the user, known as a **single-system image**. The middleware may also be responsible for providing high availability, by means of load balancing and responding to failures in individual components. [HWAN99] lists the following as desirable cluster middleware services and functions:

- **Single entry point:** A user logs on to the cluster rather than to an individual computer.
- **Single file hierarchy:** The user sees a single hierarchy of file directories under the same root directory.
- **Single control point:** There is a default node used for cluster management and control.
- **Single virtual networking:** Any node can access any other point in the cluster, even though the actual cluster configuration may consist of multiple interconnected networks. There is a single virtual network operation.
- **Single memory space:** Distributed shared memory enables programs to share variables.

as disk drives and network cards and logical items such as logical disk volumes, TCP/IP addresses, entire applications, and databases.

message to the entire cluster, causing all members to exchange messages to verify

to participate in a number of global namespaces. The following are examples of Beowulf system software:

- **Beowulf distributed process space (BPROC):** This package allows a process ID space to span multiple nodes in a cluster environment and also provides

A cluster is a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster.

16.9 RECOMMENDED READING AND WEB SITES

[BERS96] provides a good technical discussion of the design issues involved in allocating applications to client and server and in middleware approaches; the book also discusses products and standardization efforts. [REAG00a] and [REAG00b] cover client/server computing and network design approaches for supporting client/server computing.

A good overview of middleware technology and products is [BRIT04]. [MENA05] provides a performance comparison of remote procedure calls and distributed message passing.

APPENDIX A

TOPICS IN CONCURRENCY

A.1 Mutual Exclusion: Software Approaches

Dekker's Algorithm
Peterson's Algorithm

A.2 Race Conditions and Semaphores

Problem Statement
First Attempt
Second Attempt
Third Attempt
Fourth Attempt
A Good Attempt

A.3 A Barbership Problem

An Unfair Barbershop
A Fair Barbershop

A.4 Problems

It was impossible to get a conversation going; everybody was talking too much.

—Yogi Berra

A.1 MUTUAL EXCLUSION: SOFTWARE APPROACHES

Software approaches can be implemented for concurrent processes that execute on a single processor or a multiprocessor machine with shared main memory. These approaches usually assume elementary mutual exclusion at the memory access level ([LAMP91], but see Problem A.3). That is, simultaneous accesses (reading and/or writing) to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time. Beyond this, no support in the hardware, operating system, or programming language is assumed.

Dekker's Algorithm

Dijkstra [DIJK65] reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker. Following Dijkstra, we develop the solution in stages. This approach has the advantage of illustrating many of the common bugs encountered in developing concurrent programs.

FIRST ATTEMPT As mentioned earlier, any attempt at mutual exclusion must rely on some fundamental exclusion mechanism in the hardware. The most common of these is the constraint that only one access to a memory location can be made at a time. Using this constraint, we reserve a global memory location labeled `turn`. A process (P0 or P1) wishing to execute its critical section first examines the contents of `turn`. If the value of `turn` is equal to the number of the process, then the process may proceed to its critical section. Otherwise, it is forced to wait. Our waiting process repeatedly reads the value of `turn` until it is allowed to enter its critical section. This procedure is known as **busy waiting**, or **spin waiting**, because the thwarted process can do nothing productive until it gets permission to enter its critical section. Instead, it must linger and periodically check the variable; thus it consumes processor time (busy) while waiting for its chance.

After a process has gained access to its critical section and after it has completed that section, it must update the value of `turn` to that of the other process.

In formal terms, there is a shared global variable:

```
turn = 0;
```

Figure A.1a shows the program for the two processes. This solution guarantees the mutual exclusion property but has two drawbacks. First, processes must strictly alternate in their use of their critical section; therefore, the pace of execution is dictated by the slower of the two processes. If P0 uses its critical section only once per hour but P1 would like to use its critical section at a rate of 1,000 times per hour, P1 is forced to adopt the pace of P0. A much more serious problem is that if one process fails, the other process is permanently blocked. This is true whether a process fails in its critical section or outside of it.

$A_0 \quad A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7$

```

    . . . 0 . . .
    . . . 1 . . .

        (turn != 0)           (turn != 1)
    /* do nothing */;      /* do nothing */;
/* critical section*/;   /* critical section*/;
turn = 1;                turn = 0;

```

(a) First attempt

```

    . . . 0 . . .
    . . . 1 . . .

        (flag[1])           (flag[0])
    /* do nothing */;      /* do nothing */;
flag[0] = true;          flag[1] = true;
/*critical section*/;    /* critical section*/;
flag[0] = false;         flag[1] = false;

```

(b) Second attempt

```

    . . . 0 . . .
    . . . 1 . . .

        flag[0] = true;       flag[1] = true;
        (flag[1])           (flag[0])
    /* do nothing */;      /* do nothing */;
/* critical section*/;    /* critical section*/;
flag[0] = false;          flag[1] = false;

```

(c) Third attempt

```

    . . . 0 . . .
    . . . 1 . . .

        flag[0] = true;       flag[1] = true;
        (flag[1]) {           (flag[0]) {
            flag[0] = false;   flag[1] = false;
            /*delay */;        /*delay */;
            flag[0] = true;     flag[1] = true;
        }
/*critical section*/;    /* critical section*/;
flag[0] = false;          flag[1] = false;

```

(d) Fourth attempt

Figure A.1 Mutual Exclusion Attempts

The foregoing construction is that of a **coroutine**. Coroutines are designed to be able to pass execution control back and forth between themselves (see Problem 5.2). While this is a useful structuring technique for a single process, it is inadequate to support concurrent processing.

SECOND ATTEMPT The flaw in the first attempt is that it stores the name of the process that may enter its critical section, when in fact we need state information about both processes. In effect, each process should have its own key to the critical section so that if one fails, the other can still access its critical section. To meet this requirement a Boolean vector `flag` is defined, with `flag[0]` corresponding to P0 and `flag[1]` corresponding to P1. Each process may examine the other's flag but may not alter it. When a process wishes to enter its critical section, it periodically checks the other's flag until that flag has the value `false`, indicating that the other process is not in its critical section. The checking process immediately sets its own flag to `true` and proceeds to its critical section. When it leaves its critical section, it sets its flag to `false`.

The shared global variable¹ now is

```

type boolean (false = 0; true = 1);
array flag[2] = {0, 0}

```

Figure A.1b shows the algorithm. If one process fails outside the critical section, including the flag-setting code, then the other process is not blocked. In fact, the other process can enter its critical section as often as it likes, because the flag of the other process is always `false`. However, if a process fails inside its critical section or after setting its flag to `true` just before entering its critical section, then the other process is permanently blocked.

This solution is, if anything, worse than the first attempt because it does not even guarantee mutual exclusion. Consider the following sequence:

- P0 executes the **while** statement and finds `flag[1]` set to `false`
- P1 executes the **while** statement and finds `flag[0]` set to `false`
- P0 sets `flag[0]` to `true` and enters its critical section
- P1 sets `flag[1]` to `true` and enters its critical section

Because both processes are now in their critical sections, the program is incorrect. The problem is that the proposed solution is not independent of relative process execution speeds.

THIRD ATTEMPT Because a process can change its state after the other process has checked it but before the other process can enter its critical section, the second attempt failed. Perhaps we can fix this problem with a simple interchange of two statements, as shown in Figure A.1c.

As before, if one process fails inside its critical section, including the flag-setting code controlling the critical section, then the other process is blocked, and if a process fails outside its critical section, then the other process is not blocked.

¹The `type` declaration is used here to declare a data type (`boolean`) and to assign its values.

Next, let us check that mutual exclusion is guaranteed, using the point of view of process P0. Once P0 has set `flag[0]` to `true`, P1 cannot enter its critical section until after P0 has entered and left its critical section. It could be that P1 is already in its critical section when P0 sets its flag. In that case, P0 will be blocked by the `while` statement until P1 has left its critical section. The same reasoning applies from the point of view of P1.

This guarantees mutual exclusion but creates yet another problem. If both processes set their flags to `true` before either has executed the `while` statement, then each will think that the other has entered its critical section, causing deadlock.

FOURTH ATTEMPT In the third attempt, a process sets its state without knowing the state of the other process. Deadlock occurs because each process can insist on its right to enter its critical section; there is no opportunity to back off from this position. We can try to fix this in a way that makes each process more deferential: Each process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag to defer to the other process, as shown in Figure A.1d.

This is close to a correct solution but is still flawed. Mutual exclusion is still guaranteed, using similar reasoning to that followed in the discussion of the third attempt. However, consider the following sequence of events:

```
P0 sets flag[0] to true.  
P1 sets flag[1] to true.  
P0 checks flag[1].  
P1 checks flag[0].  
P0 sets flag[0] to false.  
P1 sets flag[1] to false.  
P0 sets flag[0] to true.  
P1 sets flag[1] to true.
```

This sequence could be extended indefinitely, and neither process could enter its critical section. Strictly speaking, this is not deadlock, because any alteration in the relative speed of the two processes will break this cycle and allow one to enter the critical section. This condition is referred to as **livelock**. Recall that deadlock occurs when a set of processes wishes to enter their critical sections but no process can succeed. With livelock, there are possible sequences of executions that succeed, but it is also possible to describe one or more execution sequences in which no process ever enters its critical section.

Although the scenario just described is not likely to be sustained for very long, it is nevertheless a possible scenario. Thus we reject the fourth attempt.

A CORRECT SOLUTION We need to be able to observe the state of both processes, which is provided by the array variable `flag`. But, as the fourth attempt shows, this is not enough. We must impose an order on the activities of the two processes to avoid the problem of “mutual courtesy” that we have just observed. The variable `turn` from the first attempt can be used for this purpose; in this case

the variable indicates which process has the right to insist on entering its critical region.

We can describe this solution, referred to as Dekker's algorithm, as follows. When P0 wants to enter its critical section, it sets its flag to `true`. It then checks the flag of P1. If that is `false`, P0 may immediately enter its critical section. Otherwise, P0 consults `turn`. If it finds that `turn = 0`, then it knows that it is its turn to insist and periodically checks P1's flag. P1 will at some point note that it is its turn to defer and set its flag to `false`, allowing P0 to proceed. After P0 has used its critical section, it sets its flag to `false` to free the critical section and sets `turn` to 1 to transfer the right to insist to P1.

```

a ← flag [2];
turn;
P0()
{
    (true) {
        flag [0] = true;
        (flag [1]) {
            (turn == 1) {
                flag [0] = false;
                (turn == 1) /* do nothing */;
                flag [0] = true;
            }
        }
        /* critical section */;
        turn = 1;
        flag [0] = false;
        /* remainder */;
    }
}
P1( )
{
    (true) {
        flag [1] = true;
        (flag [0]) {
            (turn == 0) {
                flag [1] = false;
                (turn == 0) /* do nothing */;
                flag [1] = true;
            }
        }
        /* critical section */;
        turn = 0;
        flag [1] = false;
        /* remainder */;
    }
}
a ← ()
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    a      ← (P0, P1);
}

```

Figure A.2 Dekker's Algorithm

Figure A.2 provides a specification of Dekker's algorithm. The construct **parbegin** (P_1, P_2, \dots, P_n) means the following: suspend the execution of the main program; initiate concurrent execution of procedures P_1, P_2, \dots, P_n ; when all of P_1, P_2, \dots, P_n have terminated, resume the main program. A verification of Dekker's algorithm is left as an exercise (see Problem A.1).

Peterson's Algorithm

Dekker's algorithm solves the mutual exclusion problem but with a rather complex program that is difficult to follow and whose correctness is tricky to prove. Peterson [PETE81] has provided a simple, elegant solution. As before, the global array variable `flag` indicates the position of each process with respect to mutual exclusion, and the global variable `turn` resolves simultaneity conflicts. The algorithm is presented in Figure A.3.

That mutual exclusion is preserved is easily shown. Consider process P_0 . Once it has set `flag[0]` to `true`, P_1 cannot enter its critical section. If P_1 already is in its critical section, then `flag[1] = true` and P_0 is blocked from entering its critical section. On the other hand, mutual blocking is prevented. Suppose that P_0 is blocked in its **while** loop. This means that `flag[1]` is `true` and `turn = 1`. P_0 can

```

    a . flag [ 2 ];
    turn;
    P0()
    {
        (true) {
            flag [ 0 ] = true;
            turn = 1;
            (flag [ 1 ] && turn == 1) /* do nothing */;
            /* critical section */;
            flag [ 0 ] = false;
            /* remainder */;
        }
    }
    P1()
    {
        (true) {
            flag [ 1 ] = true;
            turn = 0;
            (flag [ 0 ] && turn == 0) /* do nothing */;
            /* critical section */;
            flag [ 1 ] = false;
            /* remainder */;
        }
    }
    a . r()
    {
        flag [ 0 ] = false;
        flag [ 1 ] = false;
        a . r ( P0, P1 );
    }
}

```

Figure A.3 Peterson's Algorithm for Two Processes

enter its critical section when either `flag[1]` becomes `false` or `turn` becomes 0. Now consider three exhaustive cases:

1. P1 has no interest in its critical section. This case is impossible, because it implies `flag[1] = false`.
2. P1 is waiting for its critical section. This case is also impossible, because if `turn = 1`, P1 is able to enter its critical section.
3. P1 is using its critical section repeatedly and therefore monopolizing access to it. This cannot happen, because P1 is obliged to give P0 an opportunity by setting `turn` to 0 before each attempt to enter its critical section.

Thus we have a simple solution to the mutual exclusion problem for two processes. Furthermore, Peterson's algorithm is easily generalized to the case of n processes [HOFR90].

A.2 RACE CONDITIONS AND SEMAPHORES

Although the definition of a race condition, provided in Section 5.1, seems straightforward, experience has shown that students usually have difficulty pinpointing race conditions in their programs. The purpose of this section, which is based on [CARR01],² is to step through a series of examples using semaphores that should help clarify the topic of race conditions.

Problem Statement

Assume that there are two processes, **A** and **B**, each of which consists of a number of concurrent threads. Each thread includes an infinite loop in which a message is exchanged with a thread in the other process. Each message consists of an integer placed in a shared global buffer. There are two requirements:

1. After a thread A1 of process **A** makes a message available to some thread B1 in **B**, A1 can only proceed after it receives a message from B1. Similarly, after B1 makes a message available to A1, it can only proceed after it receives a message from A1.
2. Once a thread A1 makes a message available, it must make sure that no other thread in **A** overwrites the global buffer before the message is retrieved by a thread in **B**.

In the remainder of this section, we show four attempts to implement this scheme using semaphores, each of which can result in a race condition. Finally, we show a correct solution.

²I am grateful to Professor Ching-Kuang Shene of Michigan Technological University for permission to use this example.

First Attempt

Consider this approach:

```
a = 0, b = 0;
buf_a, buf_b;

a (...) {
{
    var_a;
    ...
    (true) {
        ...
        var_a = ...;
        semSignal(b);
        semWait(a);
        buf_a = var_a;
        var_a = buf_b;
        ...
    }
}
```

```
a (...) {
{
    var_b;
    ...
    (true) {
        ...
        var_b = ...;
        semSignal(a);
        semWait(b);
        buf_b = var_b;
        var_b = buf_a;
        ...
    }
}
```

This is a simple handshaking protocol. When a thread A1 in **A** is ready to exchange messages, it sends a signal to a thread in **B** and then waits for a thread B1 in **B** to be ready. Once a signal comes back from B1, which A perceives by performing `semWait(a)`, then A1 assumes that B1 is ready and performs the exchange. B1 behaves similarly, and the exchange happens regardless of which thread is ready first.

This attempt can lead to race conditions. For example, consider the following sequence, with time going vertically down the table:

a	b	a
semSignal(b)		
semWait(a)		
		semSignal(a)
		semWait(b)
buf_a = var_a		
var_a = buf_b		
		buf_b = var_b

In the preceding sequence, A1 reaches `semWait(a)` and is blocked. B1 reaches `semWait(b)` and is not blocked, but is switched out before it can update its `buf_b`. Meanwhile, A1 executes and reads from `buf_b` before it has the intended value. At this point, `buf_b` may have a value provided previously by another thread or provided by B1 in a previous exchange. This is a race condition.

A subtler race condition can be seen if two threads in **A** and **B** are active. Consider the following sequence:

a 1	a 2	a	a 2
semSignal(b)			
semWait(a)			
		semSignal(a)	
		semWait(b)	
	semSignal(b)		
	semWait(a)		
		buf_b = var_b1	
			semSignal(a)
buf_a = var_a1			
	buf_a = var_a2		

In this sequence, threads A1 and B1 attempt to exchange messages and go through the proper semaphore signaling instructions. However, immediately after the two `semWait` signals occur (in threads A1 and B1), thread A2 runs and executes `semSignal(b)` and `semWait(a)`, which causes thread B2 to execute `semSignal(a)` to release A2 from `semWait(a)`. At this point, either A1 or A2 could update `buf_a` next, and we have a race condition. By changing the sequence of execution among the threads, we can readily find other race conditions.

Lesson Learned: When a variable is shared by multiple threads, race conditions are likely to occur unless proper mutual exclusion protection is used.

Second Attempt

For this attempt, we use a semaphore to protect the shared variable. The purpose is to ensure that access to `buf_a` and `buf_b` are mutually exclusive. The program is as follows:

```

> ↵ a      a = 0, b = 0; mutex = 1;
  ↵   buf_a, buf_b;

  ↵ a _(...)

{           ↵ a _ (...)

    ↵   var_a;           ↵   var_b;

    . . .             . . .

    (true) {           (true) {

        . . .
        var_a = . . .;   var_b = . . .;
        semSignal(b);   semSignal(a);
        semWait(a);     semWait(b);

        semWait(mutex);   semWait(mutex);
        buf_a = var_a;   buf_b = var_b;
        semSignal(mutex); semSignal(mutex);
        semSignal(b);   semSignal(a);
        semWait(a);     semWait(b);

        semWait(mutex);   semWait(mutex);
        var_a = buf_b;   var_b = buf_a;
        semSignal(mutex); semSignal(mutex);
        . . .
    }
}

```

Before a thread can exchange a message, it follows the same handshaking protocol as in the first attempt. The semaphore `mutex` protects `buf_a` and `buf_b` in an attempt to assure that update precedes reading. But the protection is not adequate. Once both threads complete the first handshaking stage, the values of semaphores `a` and `b` are both 1. There are three possibilities that could occur:

1. Two threads, say A1 and B1, complete the first handshaking and continue with the second stage of the exchange.
2. Another pair of threads starts the first stage.
3. One of the current pair will continue and exchange a message with a newcomer in the other pair.

All of these possibilities can lead to race conditions. As an example of a race condition based on the third possibility, consider the following sequence:

<code>a</code>	<code>a</code>	<code>a</code>
<code>semSignal(b)</code>		
<code>semWait(a)</code>		
		<code>semSignal(a)</code>
		<code>semWait(b)</code>
<code>buf_a = var_a1</code>		
		<code>buf_b = var_b1</code>
	<code>semSignal(b)</code>	
	<code>semWait(a)</code>	
		<code>semSignal(a)</code>
		<code>semWait(b)</code>
	<code>buf_a = var_a2</code>	

In this example, after A1 and B1 go through the first handshake, they both update the corresponding global buffers. Then A2 initiates the first handshaking stage. Following this, B1 initiates the second handshaking stage. At this point, A2 updates `buf_a` before B1 can retrieve the value placed in `buf_a` by A1. This is a race condition.

Lesson Learned: Protecting a single variable may be insufficient if the use of that variable is part of a long execution sequence. Protect the whole execution sequence.

Third Attempt

For this attempt, we want to expand the critical section to include the entire message exchange (two threads each update one of two buffers and read from the other buffer). A single semaphore is insufficient because this could lead to deadlock, with each side waiting on the other. The program is as follows:

```

> ra      already = 1, adone = 0, bready = 1 bdone = 0;
` buf_a, buf_b;

a _ b(...)
{
    ` var_a;
    ...
        (true) {
    ...
        var_a =....;
        semWait(already);
        buf_a = var_a;
        semSignal(adone);
        semWait(bdone);
        var_a = buf_b;
        semSignal(already);
    ...
}
}

```

```

a _ (...)

{
    ` var_b;
    ...
        (true) {
    ...
        var_b =....;
        semWait(bready);
        buf_b = var_b;
        semSignal(bdone);
        semWait(adone);
        var_b = buf_a;
        semSignal(bready);
    ...
}
}

```

The semaphore `already` is intended to insure that no other thread in **A** can update `buf_a` while one thread from **A** enters its critical section. The semaphore `adone` is intended to insure that no thread from **B** will attempt to read `buf_a` until `buf_a` has been updated. The same considerations apply to `bready` and `bdone`. However, this scheme does not prevent race conditions. Consider the following sequence:

a ^b	a
buf_a = var_a	
semSignal(adone)	
semWait(bdone)	
	buf_b = var_b
	semSignal(bdone)
	semWait(adone)
var_a = buf_b;	
semSignal(already)	
...loop back...	
semWait(already)	
buf_a = var_a	
	var_b = buf_a

In this sequence, both A1 and B1 enter their critical sections, deposit their messages, and reach the second wait. Then A1 copies the message from B1 and leaves its critical section. At this point, A1 could loop back in its program, generate a new message, and deposit it in `buf_a`, as shown in the preceding execution sequence. Another possibility is that at this same point another thread of **A** could generate a message and put it in `buf_a`. In either case, a message is lost and a race condition occurs.

Lesson Learned: If we have a number of cooperating thread groups, mutual exclusion guaranteed for one group may not prevent interference from threads in other groups. Further, if a critical section is repeatedly entered by one thread, then the timing of the cooperation between threads must be managed properly.

Fourth Attempt

The third attempt fails to force a thread to remain in its critical section until the other thread retrieves the message. Here is an attempt to achieve this objective:

<pre> a already = 1, adone = 0, bready = 1 bdone = 0; ↵ buf_a, buf_b; a _ (...) { ↵ var_a; ... (true) { ... var_a = ...; semWait(bready); buf_a = var_a; semSignal(adone); semWait(bdone); var_a = buf_b; semSignal(aready); ... } } </pre>	<pre> a _ (...) { ↵ var_b; ... (true) { ... var_b = ...; semWait(aready); buf_b = var_b; semSignal(bdone); semWait(adone); var_b = buf_a; semSignal(bready); ... } } </pre>
---	---

In this case, the first thread in **A** to enter its critical section decrements `bready` to 0. No subsequent thread from **A** can attempt a message exchange until a thread from **B** completes the message exchange and increments `bready` to 1. This approach too can lead to race conditions, such as in the following sequence:

a ↴	a ↴2	a
semWait(bready)		
buf_a = var_a1		
semSignal(adone)		
		semWait(aready)
		buf_b = var_b1
		semSignal(bdone)
		semWait(adone)
		var_b = buf_a
		semSignal(bready)
		semWait(bready)
		...
		semWait(bdone)
		var_a2 = buf_b

In this sequence, threads A1 and B1 enter corresponding critical sections in order to exchange messages. Thread B1 retrieves its message and signals `bready`. This enables another thread from **A**, A2, to enter its critical section. If A2 is faster than A1, then A2 may retrieve the message that was intended for A1.

Lesson Learned: If the semaphore for mutual exclusion is not released by its owner, race conditions can occur. In this fourth attempt, a semaphore is locked by a thread in **A** and then unlocked by a thread in **B**. This is risky programming practice.

A Good Attempt

The reader may notice that the problem in this section is a variation of the bounded-buffer problem and can be approached in a manner similar to the discussion in Section 5.4. The most straightforward approach is to use two buffers, one for B-to-A messages and one for A-to-B messages. The size of each buffer needs to be one. To see the reason for this, consider that there is no ordering assumption for releasing threads from a synchronization primitive. If a buffer has more than one slot, then we cannot guarantee that the messages will be properly matched. For example, B1 could receive a message from A1 and then send a message to A1. But if the buffer has multiple slots, another thread in **A** may retrieve the message from the slot intended for A1.

Using the same basic approach as was used in Section 5.4, we can develop the following program:

<pre> > r_a notFull_A = 1, notFull_B = 1; > r_a notEmpty_A = 0, notEmpty_B = 0; buf_a, buf_b; a _ (...) { var_a; ... (true) { ... var_a =; semWait(notFull_A); buf_a = var_a; semSignal(notEmpty_A); semWait(notEmpty_B); var_a = buf_b; semSignal(notFull_B); ... } } </pre>	<pre> a _ (...) { var_b; ... (true) { ... var_b =; semWait(notFull_B); buf_b = var_b; semSignal(notEmpty_B); semWait(notEmpty_A); var_b = buf_a; semSignal(notFull_A); ... } } </pre>
---	--

To verify that this solution works, we need to address three issues:

1. The message exchange section is mutually exclusive within the thread group. Because the initial value of `notFull_A` is 1, only one thread in **A** can pass through `semWait(notFull_A)` until the exchange is complete as signaled

by a thread in **B** that executes `semSignal(notFull_A)`. A similar reasoning applies to threads in **B**. Thus, this condition is satisfied.

2. Once two threads enter their critical sections, they exchange messages without interference from any other threads. No other thread in **A** can enter its critical section until the thread in **B** is completely done with the exchange, and no other thread in **B** can enter its critical section until the thread in **A** is completely done with the exchange. Thus, this condition is satisfied.
3. After one thread exits its critical section, no thread in the same group can rush in and ruin the existing message. This condition is satisfied because a one-slot buffer is used in each direction. Once a thread in **A** has executed `semWait(notFull_A)` and entered its critical section, no other thread in **A** can update `buf_a` until the corresponding thread in **B** has retrieved the value in `buf_a` and issued a `semSignal(notFull_A)`.

Lesson Learned: It is well to review the solutions to well-known problems, because a correct solution to the problem at hand may be a variation of a solution to a known problem.

A.3 A BARBERSHIP PROBLEM

As another example of the use of semaphores to implement concurrency, we consider a simple barbershop problem.³ This example is instructive because the problems encountered when attempting to provide tailored access to barbershop resources are similar to those encountered in a real operating system.

Our barbershop has three chairs, three barbers, and a waiting area that can accommodate four customers on a sofa and that has standing room for additional customers (Figure A.4). Fire codes limit the total number of customers in the shop

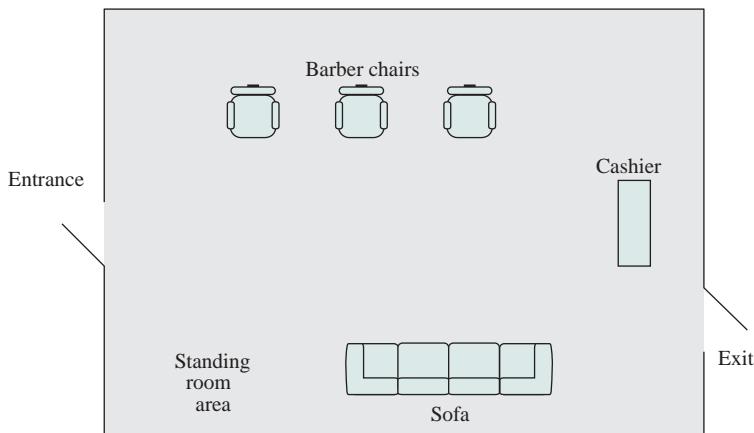


Figure A.4 The Barbershop

³I am indebted to Professor Ralph Hilzer of California State University at Chico for supplying this treatment of the problem.

to 20. In this example, we assume that the barbershop will eventually process 50 customers.

A customer will not enter the shop if it is filled to capacity with other customers. Once inside, the customer takes a seat on the sofa or stands if the sofa is filled. When a barber is free, the customer that has been on the sofa the longest is served and, if there are any standing customers, the one that has been in the shop the longest takes a seat on the sofa. When a customer's haircut is finished, any barber can accept payment, but because there is only one cash register, payment is accepted for one customer at a time. The barbers divide their time among cutting hair, accepting payment, and sleeping in their chair waiting for a customer.

An Unfair Barbershop

Figure A.5 shows an implementation using semaphores; the three procedures are listed side-by-side to conserve space. We assume that all semaphore queues are handled with a first-in-first-out policy.

The main body of the program activates 50 customers, 3 barbers, and the cashier process. We now consider the purpose and positioning of the various synchronization operators:

- **Shop and sofa capacity:** The capacity of the shop and the capacity of the sofa are governed by the semaphores `max_capacity` and `sofa`, respectively. Every time a customer attempts to enter the shop, the `max_capacity` semaphore is decremented by 1; every time a customer leaves, the semaphore is incremented. If a customer finds the shop full, then that customer's process is blocked on `max_capacity` by the `semWait` function. Similarly, the `semWait` and `semSignal` operations surround the actions of sitting on and getting up from the sofa.
- **Barber chair capacity:** There are three barber chairs, and care must be taken that they are used properly. The semaphore `barber_chair` assures that no more than three customers attempt to obtain service at a time, trying to avoid the undignified occurrence of one customer sitting on the lap of another. A customer will not get up from the sofa until at least one chair is free [`semWait(barber_chair)`], and each barber signals when a customer has left that barber's chair [`semSignal(barber_chair)`]. Fair access to the barber chairs is guaranteed by the semaphore queue organization: The first customer to be blocked is the first one allowed into an available chair. Note that, in the customer procedure, if `semWait(barber_chair)` occurred after `semSignal(sofa)`, each customer would only briefly sit on the sofa and then stand in line at the barber chairs, creating congestion and leaving the barbers with little elbow room.
- **Ensuring customers are in barber chair:** The semaphore `cust_ready` provides a wakeup signal for a sleeping barber, indicating that a customer has just taken a chair. Without this semaphore, a barber would never sleep but would begin cutting hair as soon as a customer left the chair; if no new customer had grabbed the seat, the barber would be cutting air.

```

/*
 *      a_barbershop1 */
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment = 0, receipt = 0;

! customer()
{
    semWait(max_capacity);
    enter_shop();
    semWait(sofa);
    sit_on_sofa();
    semWait(barber_chair);
    get_up_from_sofa();
    semSignal(sofa);
    sit_in_barber_chair();
    semSignal(cust_ready);
    semWait(finished);
    leave_barber_chair();
    semSignal(leave_b_chair);
    pay();
    semSignal(payment);
    semWait(receipt);
    exit_shop();
    semSignal(max_capacity)
}

! a_r()
{
    a_r (customer,...50 times,...customer, barber, barber, barber, cashier);
}

! barber()
{
    (true)
    {
        semWait(cust_ready);
        semWait(coord);
        cut_hair();
        semSignal(coord);
        semSignal(finished);
        semWait(leave_b_chair);
        semSignal(barber_chair);
    }
}

! cashier()
{
    (true)
    {
        semWait(payment);
        semWait(coord);
        accept_pay();
        semSignal(coord);
        semSignal(receipt);
    }
}

```

Figure A.5 An Unfair Barbershop

- **Holding customers in barber chair:** Once seated, a customer remains in the chair until the barber gives the signal that the haircut is complete, using the semaphore `finished`.
- **Limiting one customer to a barber chair:** The semaphore `barber_chair` is intended to limit the number of customers in barber chairs to three. However, by itself, `barber_chair` does not succeed in doing this. A customer that fails to get the processor immediately after his barber executes `semSignal(finished)` (i.e., one who falls into a trance or stops to chat with a neighbor) may still be in the chair when the next customer is given the go ahead to be seated. The semaphore `leave_b_chair` is intended to correct this problem by restraining the barber from inviting a new customer into the chair until the lingering one has announced his departure from it. In the problems at the end of this chapter, we will find that even this precaution fails to stop the mettlesome customer lap sittings.
- **Paying and receiving:** Naturally, we want to be careful when dealing with money. The cashier wants to be assured that each customer pays before leaving the shop, and the customer wants verification that payment was received (a receipt). This is accomplished, in effect, by a face-to-face transfer of the money. Each customer, upon arising from a barber chair, pays, then alerts the cashier that money has been passed over [`semSignal(payment)`], and then waits for a receipt [`semWait(receipt)`]. The cashier process repeatedly takes payments: It waits for a payment to be signaled, accepts the money, and then signals acceptance of the money. Several programming errors need to be avoided here. If `semSignal(payment)` occurred just before the action `pay`, then a customer could be interrupted after signaling; this would leave the cashier free to accept payment even though none had been offered. An even more serious error would be to reverse the positions of the `semSignal(payment)` and `semWait(receipt)` lines. This would lead to deadlock because that would cause all customers and the cashier to block at their respective `semWait` operators.
- **Coordinating barber and cashier functions:** To save money, this barbershop does not employ a separate cashier. Each barber is required to perform that task when not cutting hair. The semaphore `coord` ensures that barbers perform only one task at a time.

Table A.1 summarizes the use of each of the semaphores in the program.

The cashier process could be eliminated by merging the `payment` function into the `barber` procedure. Each barber would sequentially cut hair and then accept `pay`. However, with a single cash register, it is necessary to limit access to the `accept pay` function to one barber at a time. This could be done by treating that function as a critical section and guarding it with a semaphore.

A Fair Barbershop

Figure A.5 is a good effort, but some difficulties remain. One problem is solved in the remainder of this section; others are left as exercises for the reader (see Problem A.6).

Table A.1 Purpose of Semaphores in Figure A.5

Semaphore	Wait Operation	Signal Operation
max_capacity	Customer waits for space to enter shop.	Exiting customer signals customer waiting to enter.
sofa	Customer waits for seat on sofa.	Customer leaving sofa signals customer waiting for sofa.
barber_chair	Customer waits for empty barber chair.	Barber signals when that barber's chair is empty.
cust_ready	Barber waits until a customer is in the chair.	Customer signals barber that customer is in the chair.
finished	Customer waits until his haircut is complete.	Barber signals when cutting hair of this customer is done.
leave_b_chair	Barber waits until customer gets up from the chair.	Customer signals barber when customer gets up from chair.
payment	Cashier waits for a customer to pay.	Customer signals cashier that he has paid.
receipt	Customer waits for a receipt for payment.	Cashier signals that payment has been accepted.
coord	Wait for a barber resource to be free to perform either the hair cutting or cashiering function.	Signal that a barber resource is free.

There is a timing problem in Figure A.5 that could lead to unfair treatment of customers. Suppose that three customers are currently seated in the three barber chairs. In that case, the customers would most likely be blocked on `semWait(finished)`, and due to the queue organization they would be released in the order they entered the barber chair. However, what if one of the barbers is very fast or one of the customers is quite bald? Releasing the first customer to enter the chair could result in a situation where one customer is summarily ejected from his seat and forced to pay full price for a partial haircut while another is restrained from leaving his chair even though his haircut is complete.

The problem is solved with more semaphores, as shown in . We assign a unique customer number to each customer; this is equivalent to having each customer take a number upon entering the shop. The semaphore `mutex1` protects access to the global variable `count` so that each customer receives a unique number. The semaphore `finished` is redefined to be an array of 50 semaphores. Once a customer is seated in a barber chair, he executes `semWait(finished[custnr])` to wait on his own unique semaphore; when the barber is finished with that customer, the barber executes `semSignal(finished[b_cust])` to release the correct customer.

It remains to say how a customer's number is known to the barber. A customer places his number on the queue `enqueue1` just prior to signaling the barber with the semaphore `cust_ready`. When a barber is ready to cut hair, `dequeue1(b_cust)` removes the top customer number from `queue1` and places it in the barber's local variable `b_cust`.

```

/*
 *      a_r barbershop2 */
a_r
    max_capacity = 20;
    sofa = 4;
    barber_chair = 3, coord = 3;
    mutex1 = 1, mutex2 = 1;
    cust_ready = 0, leave_b_chair = 0, payment= 0, receipt = 0;
    finished [50] = {0};
    count;

    customer ()
    {
        custnr;
        semWait(max_capacity);
        enter_shop();
        semWait(mutex1);
        custnr = count;
        count++;
        semSignal(mutex1);
        semWait(sofa);
        sit_on_sofa();
        semWait(barber_chair);
        get_up_from_sofa();
        semSignal(sofa);
        sit_in_barber_chair();
        semWait(mutex2);
        enqueue1(custnr);
        semSignal(cust_ready);
        semSignal(mutex2);
        semWait(finished[custnr]);
        leave_barber_chair();
        semSignal(leave_b_chair);
        pay();
        semSignal(payment);
        semWait(receipt);
        exit_shop();
        semSignal(max_capacity)
    }

    main()
    {
        count := 0;
        a      (customer,...50 times,...customer, barber, barber, barber, cashier);
    }
}

    barber()
    {
        b_cust;
        (true)
        {
            semWait(cust_ready);
            semWait(mutex2);
            dequeuel(b_cust);
            semSignal(mutex2);
            semWait(coord);
            cut_hair();
            semSignal(coord);
            semSignal(finished[b_cust]);
            semWait(leave_b_chair);
            semSignal(barber_chair);
        }
    }

    cashier()
    {
        (true)
        {
            semWait(payment);
            semWait(coord);
            accept_pay();
            semSignal(coord);
            semSignal(receipt);
        }
    }
}

```

Figure A.6 A Fair Barbershop

A.4 PROBLEMS

- A.1** Demonstrate the correctness of Dekker's algorithm.

- a. Show that mutual exclusion is enforced. *Hint:* Show that when P_i enters its critical section, the following expression is true:

$$\text{flag}[i] \text{ AND } (\neg \text{flag}[1 - i])$$

- b. Show that a process requiring access to its critical section will not be delayed indefinitely. *Hint:* Consider the following cases: (1) A single process is attempting to enter the critical section; (2) both processes are attempting to enter the critical section, and (2a) $\text{turn} = 0$ and $\text{flag}[0] = \text{false}$, and (2b) $\text{turn} = 0$ and $\text{flag}[0] = \text{true}$.

- A.2** Consider Dekker's algorithm, written for an arbitrary number of processes by changing the statement executed when leaving the critical section from

$$\text{turn} = 1 - i \quad /* \text{ i.e. } P_0 \text{ sets turn to 1 and } P_1 \text{ sets turn to 0 */}$$

to

$$\text{turn} = (\text{turn} + 1) \% n \quad /* n = number of processes */$$

Evaluate the algorithm when the number of concurrently executing processes is greater than two.

- A.3** Demonstrate that the following software approaches to mutual exclusion do not depend on elementary mutual exclusion at the memory access level:

- a. the bakery algorithm
b. Peterson's algorithm

- A.4** Answer the following questions relating to the fair barbershop (Figure A.6):

- a. Does the code require that the barber who finishes a customer's haircut collect that customer's payment?
b. Do barbers always use the same barber chair?

- A.5** A number of problems remain with the fair barbershop of Figure A.6. Modify the program to correct the following problems.

- a. The cashier may accept pay from one customer and release another if two or more are waiting to pay. Fortunately, once a customer presents payment, there is no way for him to un-present it, so in the end, the right amount of money ends up in the cash register. Nevertheless, it is desirable to release the right customer as soon as his payment is taken.
b. The semaphore `leave_b_chair` supposedly prevents multiple access to a single barber chair. Unfortunately, this semaphore does not succeed in all cases. For example, suppose that all three barbers have finished cutting hair and are blocked at `semWait(leave_b_chair)`. Two of the customers are in an interrupted state just prior to `leave_b_chair`. The third customer leaves his chair and executes `semSignal(leave_b_chair)`. Which barber is released? Because the `leave_b_chair` queue is first in first out, the first barber that was blocked is released. Is that the barber that was cutting the signaling customer's hair? Maybe, but maybe not. If not, then a new customer will come along and sit on the lap of a customer that was just about to get up.
c. The program requires a customer first sits on the sofa even if a barber chair is empty. Granted, this is a rather minor problem, and fixing it makes code that is already a bit messy even messier. Nevertheless, give it a try.

This page intentionally left blank

APPENDIX B

PROGRAMMING AND OPERATING SYSTEM PROJECTS

B.1 OS/161

B.2 Simulations

B.3 Programming Projects

Textbook-Defined Projects

Additional Major Programming Projects

Small Programming Projects

B.4 Research Projects

B.5 Reading/Report Assignments

B.6 Writing Assignments

B.7 Discussion Topics

B.8 BACI

Analysis and observation, theory and experience must never disdain or exclude each other; on the contrary, they support each other.

—ON WAR, Carl Von Clausewitz

Many instructors believe that implementation or research projects are crucial to the clear understanding of operating system concepts. Without projects, it may be difficult for students to grasp some of the basic OS abstractions and interactions among components; a good example of a concept that many students find difficult to master is that of semaphores. Projects reinforce the concepts introduced in this book, give the student a greater appreciation of how the different pieces of an OS fit together, and can motivate students and give them confidence that they are capable of not only understanding but also implementing the details of an OS.

In this text, I have tried to present the concepts of OS internals as clearly as possible and have provided numerous homework problems to reinforce those concepts. Many instructors will wish to supplement this material with projects. This appendix provides some guidance in that regard and describes support material available at the instructor's Web site. The support material covers eight types of projects and other student exercises:

- OS/161 projects
- Simulation projects
- Programming projects
- Research projects
- Reading/report assignments
- Writing assignments
- Discussion topics
- BACI

B.1 OS/161

The **Instructor's Resource Center (IRC)** for this book provides support for using OS/161 as an active learning component.

OS/161 is an educational operating system developed at Harvard University [HOLL02]. It aims to strike a balance between giving students experience in working on a real operating system, and potentially overwhelming students with the complexity that exists in a fully-fledged operating system, such as Linux. Compared to most deployed operating systems, OS/161 is quite small (approximately 20,000 lines of code and comments), and therefore it is much easier to develop an understanding of the entire code base.

The source code distribution contains a full operating system source tree, including the kernel, libraries, various utilities (`ls`, `cat`, ...), and some test programs. OS/161 boots on the simulated machine in the same manner as a real system might boot on real hardware.

System/161 simulates a “real” machine to run OS/161 on. The machine features a MIPS R2000/R3000 CPU including an MMU, but no floating-point unit or cache. It also features simplified hardware devices hooked up to the system bus. These devices are much simpler than real hardware, and thus make it feasible for students to get their hands dirty without having to deal with the typical level of complexity of physical hardware. Using a simulator has several advantages: Unlike other software students write, buggy OS software may result in completely locking up the machine, making it difficult to debug and requiring a reboot. A simulator enables debuggers to access the machine below the software architecture level as if debugging was built into the CPU. In some senses, the simulator is similar to an in-circuit emulator (ICE) that you might find in industry, only it is implemented in software. The other major advantage is the speed of reboots. Rebooting real hardware takes minutes, and hence the development cycle can be frustratingly slow on real hardware. System/161 boots OS/161 in mere seconds.

The OS/161 and System/161 simulators can be hosted on a variety of platforms, including Unix, Linux, Mac OS X, and Cygwin (the free Unix environment for Windows).

The IRC includes the following:

- **Package for instructor’s Web server:** A set of html and pdf files that can be easily uploaded to the instructor’s site for the OS course, which provide all the online resources for OS/161 and S/161 access, user’s guides for students, assignments, and other useful material.
- **Getting started for instructors:** This guide lists all of the files that make up the Web site for the course and instructions on how to set up the Web site.
- **Getting started for students:** This guide explains to students step-by-step how to download and install OS/161 and S/161 on their PC.
- **Background material for students:** This consists of two documents that provide an overview of the architecture of S/161 and the internals of OS/161. These overviews are intended to be sufficient so that the student is not overwhelmed with figuring out what these systems are.
- **Student exercises:** A set of exercises that cover some of the key aspects of OS internals, including support for system calls, threading, synchronization, locks and condition variables, scheduling, virtual memory, files systems, and security

The IRC OS/161 package was prepared by Andrew Peterson and other colleagues and students at the University of Toronto.

B.2 SIMULATIONS

The IRC provides support for assigning projects based on a set of simulations developed at the University of Texas, San Antonio. Table B.1 lists the simulations by chapter. The simulators are all written in Java and can be run either locally as a Java application or online through a browser.

Table B.1 OS Simulations by Chapter

Chapter 5 – Concurrency: Mutual Exclusion and Synchronization	
Producer-consumer	Allows the user to experiment with a bounded buffer synchronization problem in the context of a single producer and a single consumer
UNIX Fork-pipe	Simulates a program consisting of pipe, dup2, close, fork, read, write, and print instructions
Chapter 6 – Concurrency: Deadlock and Starvation	
Starving philosophers	Simulates the dining philosophers problem
Chapter 8 – Virtual Memory	
Address translation	Used for exploring aspects of address translation. It supports 1- and 2-level page tables and a translation lookaside buffer
Chapter 9 – Uniprocessor Scheduling	
Process scheduling	Allows users to experiment with various process scheduling algorithms on a collection of processes and to compare such statistics as throughput and waiting time
Chapter 11 – I/O Management and Disk Scheduling	
Disk head scheduling	Supports the standard scheduling algorithms such as FCFS, SSTF, SCAN, LOOK, C-SCAN, and C-LOOK as well as double buffered versions of these
Chapter 12 – File Management	
Concurrent I/O	Simulates a program consisting of open, close, read, write, fork, wait, pthread_create, pthread_detach, and pthread_join instructions

The IRC includes the following:

1. A brief overview of the simulations available.
2. How to port them to the local environment.
3. Specific assignments to give to students, telling them specifically what they are to do and what results are expected. For each simulation, this section provides one or two original assignments that the instructor can assign to students.

These simulation assignments were developed by Adam Critchley (University of Texas at San Antonio).

B.3 PROGRAMMING PROJECTS

Three sets of programming projects are provided.

Textbook-Defined Projects

Two major programming projects, one to build a shell, or command line interpreter, and one to build a process dispatcher are described in the online portion of the

textbook. The projects can be assigned after Chapter 3 and after Chapter 9, respectively. The IRC provides further information and step-by-step exercises for developing the programs.

These projects were developed by Ian G. Graham of Griffith University, Australia.

Additional Major Programming Projects

A set of programming assignments, called machine problems (MPs), are available that are based on the Posix Programming Interface. The first of these assignments is a crash course in C, to enable the student to develop sufficient proficiency in C to be able to do the remaining assignments. The set consists of nine machine problems with different difficulty degrees. It may be advisable to assign each project to a team of two students.

Each MP includes not only a statement of the problem but a number of C files that are used in each assignment, step-by-step instructions, and a set of questions for each assignment that the student must answer that indicate a full understanding of each project. The scope of the assignments includes:

1. Create a program to run in a shell environment using basic I/O and string manipulation functions.
2. Explore and extend a simple Unix shell interpreter.
3. Modify faulty code that utilizes threads.
4. Implement a multithreaded application using thread synchronization primitives.
5. Write a user-mode thread scheduler
6. Simulate a time-sharing system by using signals and timers
7. A six-week project aimed at creating a simple yet functional networked file system. Covers I/O and file system concepts, memory management, and networking primitives.

The IRC provides specific instructions for setting up the appropriate support files on the instructor's Web site or local server.

These project assignments were developed at the University of Illinois at Urbana-Champaign, Department of Computer Science and adapted by Matt Sparks (University of Illinois at Urbana-Champaign) for use with this textbook.

Small Programming Projects

The instructor can also assign a number of small programming projects described in the IRC. The projects can be programmed by the students on any available computer and in any appropriate language: They are platform and language independent.

These small projects have certain advantages over the larger projects. Larger projects usually give students more of a sense of achievement, but students with less ability or fewer organizational skills can be left behind. Larger projects usually elicit more overall effort from the best students. Smaller projects can have a higher concepts-to-code ratio, and because more of them can be assigned, the opportunity exists to address a variety of different areas. Accordingly, the instructor's IRC

contains a series of small projects, each intended to be completed in a week or so, which can be very satisfying to both student and teacher. These projects were developed at Worcester Polytechnic Institute by Stephen Taylor, who has used and refined the projects in the course of teaching operating systems a dozen times.

B.4 RESEARCH PROJECTS

An effective way of reinforcing basic concepts from the course and for teaching students research skills is to assign a research project. Such a project could involve a literature search as well as a Web search of vendor products, research lab activities, and standardization efforts. Projects could be assigned to teams or, for smaller projects, to individuals. In any case, it is best to require some sort of project proposal early in the term, giving the instructor time to evaluate the proposal for appropriate topic and appropriate level of effort. Student handouts for research projects should include

- A format for the proposal
- A format for the final report
- A schedule with intermediate and final deadlines
- A list of possible project topics

The students can select one of the listed topics or devise their own comparable project. The IRC includes a suggested format for the proposal and final report as well as a list of possible research topics developed by Professor Tan N. Nguyen of George Mason University.

B.5 READING/REPORT ASSIGNMENTS

Another excellent way to reinforce concepts from the course and to give students research experience is to assign papers from the literature to be read and analyzed. The IRC includes a suggested list of papers to be assigned, organized by chapter. The Premium Content Web site provides a copy of each of the papers. The IRC also includes a suggested assignment wording.

B.6 WRITING ASSIGNMENTS

Writing assignments can have a powerful multiplier effect in the learning process in a technical discipline such as OS internals. Adherents of the Writing Across the Curriculum (WAC) movement (<http://wac.colostate.edu/>) report substantial benefits of writing assignments in facilitating learning. Writing assignments lead to more detailed and complete thinking about a particular topic. In addition, writing assignments help to overcome the tendency of students to pursue a subject with a minimum of personal engagement, just learning facts and problem-solving techniques without obtaining a deep understanding of the subject matter.

The IRC contains a number of suggested writing assignments, organized by chapter. Instructors may ultimately find that this is an important part of their approach to teaching the material. I would greatly appreciate any feedback on this area and any suggestions for additional writing assignments.

B.7 DISCUSSION TOPICS

One way to provide a collaborative experience is discussion topics, a number of which are included in the IRC. Each topic relates to material in the book. The instructor can set it up so that students can discuss a topic either in a class setting, an online chat room, or a message board. Again, I would greatly appreciate any feedback on this area and any suggestions for additional discussion topics.

B.8 BACI

In addition to all of the support provided at the IRC, the Ben-Ari Concurrent Interpreter (BACI) is a publicly available package that instructors may wish to use. BACI simulates concurrent process execution and supports binary and counting semaphores and monitors. BACI is accompanied by a number of project assignments to be used to reinforce concurrency concepts.

Appendix O provides a more detailed introduction to BACI, with information

This page intentionally left blank

binary semaphore A semaphore that takes on only the values 0 and 1. A binary semaphore allows only one process or thread to have access to a shared critical resource at a time.

block (1) A collection of contiguous records that are recorded as a unit; the units are separated by interblock gaps. (2) A group of bits that are transmitted as a unit.

B-tree A technique for organizing indexes. In order to keep access time to a minimum, it stores the data keys in a balanced hierarchy that continually realigns itself as items are inserted and deleted. Thus, all nodes always have a similar number of keys.

busy waiting The state of a process when it is waiting for an event but is still using CPU time.

deadlock (1) An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar wait state. (2) An impasse that occurs when multiple processes are waiting for an action by or a response from another process that is in a similar wait state.

deadlock avoidance A dynamic technique that examines each new resource request for deadlock. If the new request could lead to a deadlock, then the request is denied.

deadlock detection A technique in which requested resources are always granted when available. Periodically, the oper-

express statements in a job that are used to identify the job or to describe

execute in a different mode (kernel or process). When the mode switches from process to kernel, the program counter, processor status word, and other registers are saved. When the mode switches from kernel to process, this information is restored.

monitor A programming language construct that encapsulates r, processor status word,

medium in the same sequence as the data are ordered, or to obtain data in the same order as they were entered.

sequential file A file in which records are ordered according to the values of one or more key fields and processed in the same sequence from the beginning of the file.

server

time sharing

may be stored, transmitted, or operated on within a given computer. Typically, if a processor has a fixed-length instruc-

ABBREVIATIONS

- BOVE06** Bovet, D., and Cesati, M. *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly, 2006.
- BREN89** Brent, R. "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation."

- CARR89** Carriero, N., and Gelernter, D. "How to Write Parallel Programs: A Guide for the Perplexed." *ACM Computing Surveys*, September 1989.
- CARR01**

- CONW63** Conway, M. "Design of a Separable Transition-Diagram Compiler." *Communications of the ACM*, July 1963.
- CONW67** Conway, R., Maxwell, W., and Miller, L. *MIT Project 28.933 -0667 TD*(CONW0()31)

- DENN87** Denning, D. "An Intrusion-Detection Model." *IEEE Transactions on Software Engineering*, February 1987.
- DIJK65** Dijkstra, E. *Cooperating Sequential Processes*. Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996.) Also reprinted in [BRIN01].
- DIJK68** Dijkstra, E. "The Structure of 'THE' Multiprogramming System." *Communications of the ACM*, May 1968. Reprinted in [BRIN01].
- DIJK71** Dijkstra, E. "Hierarchical Ordering of sequential Processes." *Acta informatica*, Vol. 1, No. 2, 1971. Reprinted in [BRIN01].
- DIMI98** Dimitoglou, G. "Deadlocks and Methods for Their Detection, Prevention, and Recovery in Modern Operating Systems." *Operating Systems Review*, July 1998.
- DONA01** Donahoo, M., and Clavert, K. *The Pocket Guide to TCP/IP Sockets*. San Francisco, CA: Morgan Kaufmann, 2001.
- DOUG89** Douglas, F., and Ousterhout, J. "Process Migration in Sprite: A Status Report." *Newsletter of the IEEE Computer Society Technical 9 -12(kets)o69(Sb)eReport.DON*

- FIDG96** Fidge, C. "Fundamentals of Distributed System Observation." *IEEE Software*, November 1996.
- FINK88** Finkel, R. *An Operating Systems Vade Mecum*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- FINK89** Finkel, R. "The Process Migration Mechanism of Charlotte." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*

730

GOYE99

KLEI75

Kleinrock, L. *Queueing Systems, Volume I: Theory*. New York: Wiley, 1975.

KLEI76

Kleinrock, L. *Queueing Systems, Volume II: Computer Applications*. New York: Wiley, 1976.

KLEI95

- LEON07** Leonard, T. "Dragged Kicking and Screaming: Source Multicore." *Proceedings, Game Developers Conference 2007*, March 2007.
- LERO76** Leroudier, J., and Potier, D. "Principles of Optimality for Multiprogramming."

LURI94 Lurie, D., and Moore, R.

- NACH97** Nachenberg, C. "Computer Virus-Antivirus Coevolution." *Communications of*

SHA90

Sha, L.; Rajkumar, R.; and Lehoczky, J. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*,

- ZAH090** Zahorjan, J., and McCann, C. "Processor Scheduling in Shared Memory Multiprocessors." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1990.
- ZAJC93** Zajcew, R., et al. "An OSF/1 UNIX for Massively Parallel Multicomputers." *Proceedings, W9l8*



permanent, 259
record, 541–543
Block operation, 162

middleware, 687–689
network for, 679
servers in, 678–679
terminology of, 679
Client-server model, 85–86
Clock algorithm, 367, 381, 386



Electronic mail facility, 629
Elevator scheduler, 510

security (*See* File system security)
UNIX, 553–560
File object, Linux, 514, 562, 564
File organization/access, 527–532
 criteria for, 527–528
 direct file, 532
 hash file, 532
 indexed file, 531–532
 indexed sequential file, 530–531
 performance, grades of, 529
 pile, 529–530
 sequential file, 530
 types of, common, 528–529
Files, 522
 allocation (*See* File allocation)
 cache consistency, 686–687
 closing, 522
 creation of, 522
 deletion of, 522
 direct, 532
 directories (*See* File directories)
 field, input/output, 522–523
 hashed, 532
 indexed, 531–532
 indexed sequential, 530–531

**G**

Gang scheduling, 439
GCC (GNU Compiler Collection), 665
Generality, 481, 661
General message format, 236–237
General semaphores, 215, 222
Generic_all access bits, 671
Generic decryption (GD), 657–658
Generic_execute access bits, 671
Generic_read access bits, 671
Generic_write access bits, 671
Global coverage, 661
Global replacement policy, 371
Global scope, 372
Global state, 18-13
Grand Central Dispatch (GCD), 78–79
Granularity, 432–433. *See also* Parallelism
Graphical Processing Units (GPUs), 10
Graphic user interface (GUI), 681
Group
 concept of, 705
 resources, 705
 SIDs, 669
Guard pages, 667
Gupta Corp, 684

H

Hackers, 616–618
Hamming code, 500

system, 135



Ndeps, 97
Nearest fit strategy, 544
NEC V8xx, 579
Negatively correlated, 19-13
Negotiation of process migration, 18-7-9
Nested task flag, 132
.NET framework of, 75
Network access layer, 17-7-8
Network-based IDS, 144-145
Network operating system, 17-3
Network protocols, 17-1-23
 architecture of, 17-3-6
 definition of, 17-4
 elements of, 17-4-5
 Linux networking, 17-21-22
 sockets, 17-15-20
 tasks performed for, 17-3-4
 TCP/IP protocol architecture, 17-6-15
Networks, 615-616, 679. *See also*
 specific networks

avoidance approaches for, 265
central themes of, 199
commercial, 578
concurrency, concerns of, 204
development of, 62–63
distributed, 73
eCos (*See* Embedded Configurable Operating System (eCos))
embedded (*See* Embedded operating systems)
evolution of, 52–62
functions, 48–52
information in, protection and security of, 68–69
interfaces of, typical, 50
Linux (*See* Linux)
Mac OS X, 94
memory management in, 66–68
Microsoft (*See* Microsoft Windows)
modern, development leading to, 71–73
multiprocessor/multicore, 77–79
objectives/functions of, 48–52
organization of, 82–84
overview of, 46–101
process-based, 142–143
processes, 62–66, 140–143
real-time, 443–447, 577
resource management in, 50–51, 69–70
services provided by, 49
structure, 200



resources, 278
scheduling, 404, 435–436
security issues, 143–147
with smallest resident set, 378
spawning, 115
state of (*See* Process state)
state transitions, 396
structure, 183
suspension, 378–379
switching, 137–139
synchronization, 432
table entry, 150
tables, 127–128
termination of, 115–116
threads and, 66, 158–164, 169436



- highest ratio next, 413
- round robin, 407–410
- shortest process next, 410–411
- shortest remaining time, 411–413
- Scount value, 219
- Search operation, 537
- Secondary memory, 27
- Secondary storage management, 543–551
 - file allocation, 543–547
 - free space management, 547–550
 - reliability, 551
 - volumes, 550
- Second moment, [19-9](#)
- Second-order statistics, [19-15–17](#)
- Sector, 565
- Security Descriptor (SD), 87
- Security ID (SID), 668
- Security reference monitor, 84
- Security Requirements for Cryptographic Modules*, 651

T

Tape drives, 509
 Tasks, 598. *See also* Process/processes
 aperiodic, 451
 deadline scheduling for, 448–451
 hard real-time, 443
 Linux, 186–188
 periodic, 450
 real-time, Linux, 461
 soft real-time, 443
 TCP header, 17-9
 TCP segment, 17-14
 TCP/IP protocol architecture, 17-6–15
 applications of, 17-14–15
 concepts of, 17-12
 Internet Protocol (IP), and IPv6, 17-10–11
 layers, 17-7–8
 Linux kernel components of, 17-20
 operation of, 17-12–14
 protocol data units (PDUs) in, 17-13
 User Datagram Protocol (UDP) and, 17-8–10
 TELNET, 17-15
 Temporal locality, 42
 Terminals, 509
 Termination of process states, 180
 Textbook-defined projects, B-4–B-5
 Thin client, 685
 Thrashing, load control, 344
 Threading granularity options, 174
 Threads, 66, 157–192, 585. *See also* specific types of
 benefits of, 161
 bottom-half kernel, 463
 execution state, 160
 functionality of, 162–164
 interactive, 464
 kernel-level (KLT), 168–169, 182
 Linux process and, management of, 186–189
 MAC OS Grand Central Dispatch (GCD), 189–192
 management of, 186–189
 many-to-many relationships of, 169–170
 Microsoft Windows 7, improvements in, 89
 migration, 466
 multithreaded process models, 160
 multithreading, 159–162, 171–175
 objects, 177–179
 one-to-many relationships of, 170–171
 operations associated with change in, 162
 priorities, 466–468
 processes and, 158–164, 169, 182
 process operation latencies (μ s), 168
 processor affinity, 177
 remote procedure call (RPC), 1683
 mnemonic conventions for system calls, 161–162,
 for aSMP, 486



Tree representation of buddy system, 319
Tree-structured file directory, 538, 539
Trial, 19-5
Trigger, 623
Triggering phase, 624
Trivial File Transfer Protocol (TFTP), 17-5
TRIX, 170
Trojan horse, 620, 621-622
Turnaround time (TAT), 401, 404, 419
Two-handed clock algorithm, 381
Two-level hierarchical page table, 348
Two-level memory
 characteristics of, 39-45
 locality, 40-42
 operation of, 42
 performance of, 25-26, 42-45
Two-priority categories, 416
Two-state process model, 112-114

U

U area, 150-151
ULT. *See User-level threads (ULT)*
Ultrafast spreading, 630
Unauthorized disclosure, 610
Unblock state, 162u69()-250(1a2506fio0k)-25.9(,)54.9()-250(162u691e)
T modelcontro50(memorlsur9(,)54.9()-250(4165593)5600-1.1a2506cal)

Variable-allocation replacement policy, 370
global scope, 372
local scope, 372–376
Variable-interval sampled working set (VSWS) policy, 375–376
Variable-length spanned, 541
Variable-length unspanned, 542
Variable priority classes, 467
Variant, 582
Variance, 19-9
VAX/VMS, 80
Venn diagrams, 19-4
Verification step of authentication, 145
Very coarse-grained parallelism, 432–433
VFS. *See*



Win32, 84
Windowing/graphics system, 83
Windows. *See Microsoft Windows*