CHAPTER 1

Introduction

Chapter 1 introduces the general topic of operating systems and a handful of important concepts (multiprogramming, time sharing, distributed system, and so on). The purpose is to show *why* operating systems are what they are by showing *how* they developed. In operating systems, as in much of computer science, we are led to the present by the paths we took in the past, and we can better understand both the present and the future by understanding the past.

Additional work that might be considered is learning about the particular systems that the students will have access to at your institution. This is still just a general overview, as specific interfaces are considered in Chapter 3.

Exercises

1.13 In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.

a. What are two such problems?

b. Can we ensure the same degree of security in a time-shared machine as in a dedicated machine? Explain your answer.

Answer:

a. Stealing or copying one's programs or data; using system resources (CPU, memory, disk space, peripherals) without proper accounting.

b. Probably not, since any protection scheme devised by humans can inevitably be broken by a human, and the more complex the scheme, the more difficult it is to feel confident of its correct implementation.

1.14 The issue of resource utilization shows up in different forms in different types of operating systems. List what resources must be managed carefully in the following settings:

a. Mainframe or minicomputer systems

b. Workstations connected to servers

c. Handheld computers

Answer:

a. Mainframes: memory and CPU resources, storage, network bandwidth

b. Workstations: memory and CPU resources

c. Handheld computers: power consumption, memory resources

1.15 Under what circumstances would a user be better off using a time-sharing system rather than a PC or a single-user workstation?
Answer: When there are few other users, the task is large, and the hardware is fast, time-sharing makes sense. The full power of the system can be brought to bear on the user's problem. The problem can be solved faster than on a personal computer. Another case occurs when lots of other users need resources at the same time.

A personal computer is best when the job is small enough to be executed reasonably on it and when performance is sufficient to execute the program to the user's satisfaction.

1.16 Identify which of the functionalities listed below need to be supported by the operating system for (a) handheld devices and (b) real-time systems.

a. Batch programming

b. Virtual memory

c. Time sharing

Answer: For real-time systems, the operating system needs to support virtual memory and time sharing in a fair manner. For handheld systems, the operating system needs to provide virtual memory, but does not need to provide time-sharing. Batch programming is not necessary in both settings.

1.17 Describe the differences between symmetric and asymmetric multiprocessing. What are three advantages and one disadvantage of multiprocessor systems?
Answer: Symmetric multiprocessing treats all processors as equals, and I/O can be processed on any CPU. Asymmetric multiprocessing has one master CPU and the remainder CPUs are slaves. The master distributes tasks among the slaves, and I/O is usually done by the master only. Multiprocessors can save money by not duplicating power supplies, housings, and peripherals. They can execute programs more quickly and can have increased reliability. They are also more complex in both hardware and software than uniprocessor systems.

1.18 How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?
Answer: Clustered systems are typically constructed by combining multiple computers into a single system to perform a computational

task distributed across the cluster. Multiprocessor systems on the other hand could be a single physical entity comprising of multiple CPUs. A clustered system is less tightly coupled than a multiprocessor system. Clustered systems communicate using messages, while processors in a multiprocessor system could communicate using shared memory.

In order for two machines to provide a highly available service, the state on the two machines should be replicated and should be consistently updated. When one of the machines fails, the other could then takeover the functionality of the failed machine.

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

Answer: The client-server model firmly distinguishes the roles of the client and server. Under this model, the client requests services that are provided by the server. The peer-to-peer model doesn't have such strict roles. In fact, all nodes in the system are considered peers and thus may act as *either* clients or servers—or both. A node may request a service from another peer, or the node may in fact provide such a service to other peers in the system.

For example, let's consider a system of nodes that share cooking recipes. Under the client-server model, all recipes are stored with the server. If a client wishes to access a recipe, it must request the recipe from the specified server. Using the peer-to-peer model, a peer node could ask other peer nodes for the specified recipe. The node (or perhaps nodes) with the requested recipe could provide it to the requesting node. Notice how each peer may act as both a client (it may request recipes) and as a server (it may provide recipes).

1.20 Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and disadvantages of each.

Answer: Consider the following two alternatives: **asymmetric clustering** and **parallel clustering**. With asymmetric clustering, one host runs the database application with the other host simply monitoring it. If the server fails, the monitoring host becomes the active server. This is appropriate for providing redundancy. However, it does not utilize the potential processing power of both hosts. With parallel clustering, the database application can run in parallel on both hosts. The difficulty in implementing parallel clusters is providing some form of distributed locking mechanism for files on the shared disk.

1.21 How are network computers different from traditional personal computers? Describe some usage scenarios in which it is advantageous to use network computers.

Answer: A network computer relies on a centralized computer for most of its services. It can therefore have a minimal operating system to manage its resources. A personal computer on the other hand has to be capable of providing all of the required functionality in a stand-alone manner without relying on a centralized manner. Scenarios where administrative costs are high and where sharing leads to more efficient

use of resources are precisely those settings where network computers are preferred.

1.22 What is the purpose of interrupts? What are the differences between a trap and an interrupt? Can traps be generated intentionally by a user program? If so, for what purpose?
Answer: An interrupt is a hardware-generated change of flow within the system. An interrupt handler is summoned to deal with the cause of the interrupt; control is then returned to the interrupted context and instruction. A trap is a software-generated interrupt. An interrupt can be used to signal the completion of an I/O to obviate the need for device polling. A trap can be used to call operating system routines or to catch arithmetic errors.

1.23 Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

a. How does the CPU interface with the device to coordinate the transfer?

b. How does the CPU know when the memory operations are complete?

c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

Answer: The CPU can initiate a DMA operation by writing values into special registers that can be independently accessed by the device. The device initiates the corresponding operation once it receives a command from the CPU. When the device is finished with its operation, it interrupts the CPU to indicate the completion of the operation.

Both the device and the CPU can be accessing memory simultaneously. The memory controller provides access to the memory bus in a fair manner to these two entities. A CPU might therefore be unable to issue memory operations at peak speeds since it has to compete with the device in order to obtain access to the memory bus.

1.24 Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.
Answer: An operating system for a machine of this type would need to remain in control (or monitor mode) at all times. This could be accomplished by two methods:

a. Software interpretation of all user programs (like some BASIC, Java, and LISP systems, for example). The software interpreter would provide, in software, what the hardware does not provide.

b. Require that all programs be written in high-level languages so that all object code is compiler-produced. The compiler would generate

(either in-line or by function calls) the protection checks that the hardware is missing.

1.25 Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

Answer: Caches are useful when two or more components need to exchange data, and the components perform transfers at differing speeds. Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device. The data in the cache must be kept consistent with the data in the components. If a component has a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum. A component may be eliminated by an equal-sized cache, but only if: (a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and (b) the cache is affordable, because faster storage tends to be more expensive.

1.26 Consider an SMP system similar to what is shown in Figure 1.6. Illustrate with an example how data residing in memory could in fact have two different values in each of the local caches.

Answer: Say processor 1 reads data A with value 5 from main memory into its local cache. Similarly, processor 2 reads data A into its local cache as well. Processor 1 then updates A to 10. However, since A resides in processor 1's local cache, the update only occurs there and not in the local cache for processor 2.

1.27 Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

a. Single-processor systems

b. Multiprocessor systems

c. Distributed systems

Answer: In single-processor systems, the memory needs to be updated when a processor issues updates to cached values. These updates can be performed immediately or in a lazy manner. In a multiprocessor system, different processors might be caching the same memory location in its local caches. When updates are made, the other cached locations need to be invalidated or updated. In distributed systems, consistency of cached memory values is not an issue. However, consistency problems might arise when a client caches file data.

1.28 Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

Answer: The processor could keep track of what locations are associated with each process and limit access to locations that are outside

of a program's extent. Information regarding the extent of a program's memory could be maintained by using base and limits registers and by performing a check for every memory access.

1.29 What network configuration would best suit the following environments?

a. A dormitory floor

b. A university campus

c. A state

d. A nation

Answer:

a. **A dormitory floor**—A LAN.

b. **A university campus**—A LAN, possibly a WAN for very large campuses.

c. **A state**—A WAN.

d. **A nation**—A WAN.

1.30 Define the essential properties of the following types of operating systems:

a. Batch

b. Interactive

c. Time sharing

d. Real time

e. Network

f. SMP

g. Distributed

h. Clustered

i. Handheld

Answer:

a. **Batch**. Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off-line operation, spooling, and multiprogramming. Batch is good for executing large jobs that need little interaction; they can be submitted and picked up later.

b. **Interactive**. This system is composed of many short transactions where the results of the next transaction may be unpredictable. Response time needs to be short (seconds) since the user submits and waits for the result.

c. **Time sharing**. This system uses CPU scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another. Instead of having a job defined by spooled card images, each program reads its next control card from the terminal, and output is normally printed immediately to the screen.

d. **Real time**. Often used in a dedicated application, this system reads information from sensors and must respond within a fixed amount of time to ensure correct performance.

e. **Network**. Provides operating system features across a network such as file sharing.

f. **SMP**. Used in systems where there are multiple CPUs each running the same copy of the operating system. Communication takes place across the system bus.

g. **Distributed**.This system distributes computation among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines, such as a high-speed bus or local area network.

h. **Clustered**. A clustered system combines multiple computers into a single system to perform computational tasks distributed across the cluster.

i. **Handheld**. A small computer system that performs simple tasks such as calendars, email, and web browsing. Handheld systems differ from traditional desktop systems with smaller memory and display screens and slower processors.

1.31 What are the tradeoffs inherent in handheld computers?
Answer: Handheld computers are much smaller than traditional desktop PCs. This results in smaller memory, smaller screens, and slower processing capabilities than a standard desktop PC. Because of these limitations, most handhelds currently can perform only basic tasks such as calendars, email, and simple word processing. However, due to their small size, they are quite portable and, when they are equipped with wireless access, can provide remote access to electronic mail and the world wide web.

1.32 Identify several advantages and several disadvantages of open-source operating systems. Include the types of people who would find each aspect to be an advantage or a disadvantage.
Answer: Open source operating systems have the advantages of having many people working on them, many people debugging them, ease of access and distribution, and rapid update cycles. Further, for students and programmers there is certainly an advantage to being able to view and modify the source code. Typically open source operating systems are free for some forms of use, usually just requiring payment for support services. Commercial operating system companies usually do not like the competition that open source operating systems bring because

these features are difficult to compete against. Some open source operating systems do not offer paid support programs. Some companies avoid open source projects because they need paid support, so that they have some entity to hold accountable if there is a problem or they need help fixing an issue. Finally, some complain that a lack of discipline in the coding of open source operating systems means that backward-compatiblity is lacking making upgrades difficult, and that the frequent release cycle exacerbates these issues by forcing users to upgrade frequently.

Operating System Structures

Chapter 2 is concerned with the operating-system interfaces that users (or at least programmers) actually see: system calls. The treatment is somewhat vague since more detail requires picking a specific system to discuss. This chapter is best supplemented with exactly this detail for the specific system the students have at hand. Ideally they should study the system calls and write some programs making system calls. This chapter also ties together several important concepts including layered design, virtual machines, Java and the Java virtual machine, system design and implementation, system generation, and the policy/mechanism difference.

Exercises

2.12 The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories and discuss how they differ.

Answer: One class of services provided by an operating system is to enforce protection between different processes running concurrently in the system. Processes are allowed to access only those memory locations that are associated with their address spaces. Also, processes are not allowed to corrupt files associated with other users. A process is also not allowed to access devices directly without operating system intervention. The second class of services provided by an operating system is to provide new functionality that is not supported directly by the underlying hardware. Virtual memory and file systems are two such examples of new services provided by an operating system.

2.13 Describe three general methods for passing parameters to the operating system.

Answer:

a. Pass parameters in registers

b. Registers pass starting addresses of blocks of parameters

c. Parameters can be placed, or *pushed*, onto the *stack* by the program, and *popped* off the stack by the operating system

2.14 Describe how you could obtain a statistical profile of the amount of time spent by a program executing different sections of its code. Discuss the importance of obtaining such a statistical profile.
Answer: One could issue periodic timer interrupts and monitor what instructions or what sections of code are currently executing when the interrupts are delivered. A statistical profile of which pieces of code were active should be consistent with the time spent by the program in different sections of its code. Once such a statistical profile has been obtained, the programmer could optimize those sections of code that are consuming more of the CPU resources.

2.15 What are the five major activities of an operating system in regard to file management?
Answer:

- The creation and deletion of files
- The creation and deletion of directories
- The support of primitives for manipulating files and directories
- The mapping of files onto secondary storage
- The backup of files on stable (nonvolatile) storage media

2.16 What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?
Answer: Each device can be accessed as though it was a file in the file system. Since most of the kernel deals with devices through this file interface, it is relatively easy to add a new device driver by implementing the hardware-specific code to support this abstract file interface. Therefore, this benefits the development of both user program code, which can be written to access devices and files in the same manner, and device-driver code, which can be written to support a well-defined API. The disadvantage with using the same interface is that it might be difficult to capture the functionality of certain devices within the context of the file access API, thereby resulting in either a loss of functionality or a loss of performance. Some of this could be overcome by the use of the ioctl operation that provides a general-purpose interface for processes to invoke operations on devices.

2.17 Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?
Answer: An user should be able to develop a new command interpreter using the system-call interface provided by the operating system. The command interpreter allows an user to create and manage processes and also determine ways by which they communicate (such as through pipes and files). As all of this functionality could be accessed by an user-level program using the system calls, it should be possible for the user to develop a new command-line interpreter.

2.18 What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?
Answer: TBA

2.19 Why is the separation of mechanism and policy desirable?
Answer: Mechanism and policy must be separate to ensure that systems are easy to modify. No two system installations are the same, so each installation may want to tune the operating system to suit its needs. With mechanism and policy separate, the policy may be changed at will while the mechanism stays unchanged. This arrangement provides a more flexible system.

2.20 It is sometimes difficult to achieve a layered approach if two components of the operating system are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.
Answer: The virtual memory subsystem and the storage subsystem are typically tightly coupled and requires careful design in a layered system due to the following interactions. Many systems allow files to be mapped into the virtual memory space of an executing process. On the other hand, the virtual memory subsystem typically uses the storage system to provide the backing store for pages that do not currently reside in memory. Also, updates to the file system are sometimes buffered in physical memory before it is flushed to disk, thereby requiring careful coordination of the usage of memory between the virtual memory subsystem and the file system.

2.21 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?
Answer: Benefits typically include the following: (a) adding a new service does not require modifying the kernel, (b) it is more secure as more operations are done in user mode than in kernel mode, and (c) a simpler kernel design and functionality typically results in a more reliable operating system. User programs and system services interact in a microkernel architecture by using interprocess communication mechanisms such as messaging. These messages are conveyed by the operating system. The primary disadvantages of the microkernel architecture are the overheads associated with interprocess communication and the frequent use of the operating system's messaging functions in order to enable the user process and the system service to interact with each other.

2.22 In what ways is the modular kernel approach similar to the layered approach? In what ways does it differ from the layered approach?
Answer: The modular kernel approach requires subsystems to interact with each other through carefully constructed interfaces that are typically narrow (in terms of the functionality that is exposed to external modules). The layered kernel approach is similar in that respect. However, the layered kernel imposes a strict ordering of subsystems such that subsystems at the lower layers are not allowed to invoke opera-

tions corresponding to the upper-layer subsystems. There are no such restrictions in the modular-kernel approach, wherein modules are free to invoke each other without any constraints.

2.23 What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?
Answer: The system is easy to debug, and security problems are easy to solve. Virtual machines also provide a good platform for operating system research since many different operating systems can run on one physical system.

2.24 Why is a just-in-time compiler useful for executing Java programs?
Answer: Java is an interpreted language. This means that the JVM interprets the bytecode instructions one at a time. Typically, most interpreted environments are slower than running native binaries, for the interpretation process requires converting each instruction into native machine code. A just-in-time (JIT) compiler compiles the bytecode for a method into native machine code the first time the method is encountered. This means that the Java program is essentially running as a native application (of course, the conversion process of the JIT takes time as well, but not as much as bytecode interpretation). Furthermore, the JIT caches compiled code so that it can be reused the next time the method is encountered. A Java program that is run by a JIT rather than a traditional interpreter typically runs much faster.

2.25 What is the relationship between a guest operating system and a host operating system in a system like VMware? What factors need to be considered in choosing the host operating system?
Answer: A guest operating system provides its services by mapping them onto the functionality provided by the host operating system. A key issue that needs to be considered in choosing the host operating system is whether it is sufficiently general in terms of its system-call interface in order to be able to support the functionality associated with the guest operating system.

2.26 The experimental Synthesis operating system has an assembler incorporated within the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and to system-performance optimization.
Answer: Synthesis is impressive due to the performance it achieves through on-the-fly compilation. Unfortunately, it is difficult to debug problems within the kernel due to the fluidity of the code. Also, such compilation is system specific, making Synthesis difficult to port (a new compiler must be written for each architecture).

Processes

In this chapter we introduce the concepts of a process and concurrent execution; These concepts are at the very heart of modern operating systems. A process is a program in execution and is the unit of work in a modern time-sharing system. Such a system consists of a collection of processes: Operating-system processes executing system code and user processes executing user code. All these processes can potentially execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive. We also introduce the notion of a thread (lightweight process) and interprocess communication (IPC). Threads are discussed in more detail in Chapter 4.

Exercises

3.6 Describe the differences among short-term, medium-term, and long-term scheduling.
Answer:

a. **Short-term** (CPU scheduler)—selects from jobs in memory those jobs that are ready to execute and allocates the CPU to them.

b. **Medium-term**—used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.

c. **Long-term** (job scheduler)—determines which jobs are brought into memory for processing.

The primary difference is in the frequency of their execution. The short-term must select a new process quite often. Long-term is used much less often since it handles placing jobs in the system and may wait a while for a job to finish before it admits another one.

3.7 Describe the actions taken by a kernel to context-switch between processes.

Answer: In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

3.8 Construct a process tree similar to Figure 3.9. To obtain process information for the UNIX or Linux system, use the command `ps -ael`. Use the command `man ps` to get more information about the `ps` command. On Windows systems, you will have to use the task manager.
Answer: Answer: Results will vary widely.

3.9 Including the initial parent process, how many processes are created by the program shown in Figure 3.28?
Answer: 8 processes are created. The program online includes printf() statements to better understand how many processes have been created.

3.10 Using the program in Figure 3.29, identify the values of `pid` at lines `A`, `B`, `C`, and `D`. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)
Answer: Answer: A = 0, B = 2603, C = 2603, D = 2600

3.11 Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.
Answer: Simple communication works well with ordinary pipes. For example, assume we have a process that counts characters in a file. An ordinary pipe can be used where the producer writes the file to the pipe and the consumer reads the files and counts the number of characters in the file. Next, for an example where named pipes are more suitable, consider the situation where several processes may write messages to a log. When processes wish to write a message to the log, they write it to the named pipe. A server reads the messages from the named pipe and writes them to the log file.

3.12 Consider the RPC mechanism. Describe the undesirable circumstances that could arise from not enforcing either the "at most once" or "exactly once" semantics. Describe possible uses for a mechanism that had neither of these guarantees.
Answer: If an RPC mechanism cannot support either the "at most once" or "at least once" semantics, then the RPC server cannot guarantee that a remote procedure will not be invoked multiple occurrences. Consider if a remote procedure were withdrawing money from a bank account on a system that did not support these semantics. It is possible that a single invocation of the remote procedure might lead to multiple withdrawals on the server.

For a system to support either of these semantics generally requires the server maintain some form of client state such as the timestamp described in the text.

If a system were unable to support either of these sematics, then such a system could only safely provide remote procedures that do not

alter data or provide time-sensitive results. Using our bank account as an example, we certainly require "at most once" or "at least once" semantics for performing a withdrawal (or deposit!). However, an inquiry into an account balance or other accunt information such as name, address, etc. does not require these semantics.

3.13 Using the program shown in Figure 3.30, explain what the output will be at `Line A`.
Answer: Yhe result is still 5 as the child updates its copy of value. When control returns to the parent, its value remains at 5.

3.14 What are the benefits and detriments of each of the following? Consider both the systems and the programmers' levels.

a. Synchronous and asynchronous communication

b. Automatic and explicit buffering

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

Answer:

a. **Synchronous and asynchronous communication**—A benefit of synchronous communication is that it allows a rendezvous between the sender and receiver. A disadvantage of a blocking send is that a rendezvous may not be required and the message could be delivered asynchronously. As a result, message-passing systems often provide both forms of synchronization.

b. **Automatic and explicit buffering**—Automatic buffering provides a queue with indefinite length, thus ensuring the sender will never have to block while waiting to copy a message. There are no specifications on how automatic buffering will be provided; one scheme may reserve sufficiently large memory where much of the memory is wasted. Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely that memory will be wasted with explicit buffering.

c. **Send by copy and send by reference**—Send by copy does not allow the receiver to alter the state of the parameter; send by reference does allow it. A benefit of send by reference is that it allows the programmer to write a distributed version of a centralized application. Java's RMI provides both; however, passing a parameter by reference requires declaring the parameter as a remote object as well.

d. **Fixed-sized and variable-sized messages**—The implications of this are mostly related to buffering issues; with fixed-size messages, a buffer with a specific size can hold a known number of messages. The number of variable-sized messages that can be held by such a buffer is unknown. Consider how Windows 2000 handles this situation: with fixed-sized messages (anything < 256 bytes),

the messages are copied from the address space of the sender to the address space of the receiving process. Larger messages (i.e. variable-sized messages) use shared memory to pass the message.

Threads

The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Many modern operating systems now provide features for a process to contain multiple threads of control. This chapter introduces many concepts associated with multithreaded computer systems and covers how to use Java to create and manipulate threads. We have found it especially useful to discuss how a Java thread maps to the thread model of the host operating system.

Exercises

4.7 Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution

Answer: (1) Any kind of sequential program is not a good candidate to be threaded. An example of this is a program that calculates an individual tax return. (2) Another example is a "shell" program such as the C-shell or Korn shell. Such a program must closely monitor its own working space such as open files, environment variables, and current working directory.

4.8 Describe the actions taken by a thread library to context switch between user-level threads.

Answer: Context switching between user threads is quite similar to switching between kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread. This act typically involves saving and restoring the state of the registers.

4.9 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

Answer: When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of

performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system.

4.10 Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

Answer: The threads of a multithreaded process share heap memory and global variables. Each thread has its separate set of register values and a separate stack.

4.11 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?
Answer: A multithreaded system comprising of multiple user-level threads cannot make use of the different processors in a multiprocessor system simultaneously. The operating system sees only a single process and will not schedule the different threads of the process on separate processors. Consequently, there is no performance benefit associated with executing multiple user-level threads on a multiprocessor system.

4.12 As described in Section 4.5.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. However, many operating systems—such as Windows XP and Solaris—treat processes and threads differently. Typically, such systems use a notation wherein the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.
Answer: On one hand, in systems where processes and threads are considered as similar entities, some of the operating system code could be simplified. A scheduler, for instance, can consider the different processes and threads on an equal footing without requiring special code to examine the threads associated with a process during every scheduling step. On the other hand, this uniformity could make it harder to impose process-wide resource constraints in a direct manner. Instead, some extra complexity is required to identify which threads correspond to which process and perform the relevant accounting tasks.

4.13 The program shown in Figure 4.14 uses the Pthreads API. What would be output from the program at `LINE C` and `LINE P`?
Answer: Output at `LINE C` is 5. Output at `LINE P` is 0.

4.14 Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level

threads in the program be greater than the number of processors in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processors.

b. The number of kernel threads allocated to the program is equal to the number of processors.

c. The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user-level threads.

Answer: When the number of kernel threads is less than the number of processors, then some of the processors would remain idle since the scheduler maps only kernel threads to processors and not user-level threads to processors. When the number of kernel threads is exactly equal to the number of processors, then it is possible that all of the processors might be utilized simultaneously. However, when a kernel-thread blocks inside the kernel (due to a page fault or while invoking system calls), the corresponding processor would remain idle. When there are more kernel threads than processors, a blocked kernel thread could be swapped out in favor of another kernel thread that is ready to execute, thereby increasing the utilization of the multiprocessor system.

4.15 Write a multithreaded Java, Pthreads, or Win32 program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.
Answer: Please refer to the supporting Web site for source code solution.

4.16 Modify the socket-based date server (Figure 3.19) in Chapter 3 so that the server services each client request in a separate thread.
Answer: Please refer to the supporting Web site for source code solution.

4.17 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a multithreaded program that generates the Fibonacci series using either the Java, Pthreads, or Win32 thread library. This program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that is shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having

the parent thread wait for the child thread to finish using the techniques described in Section 4.3.
Answer: Please refer to the supporting Web site for source code solution.

4.18 Exercise 3.18 in Chapter 3 specifies designing an echo server using the Java threading API. However, this server is single-threaded, meaning the server cannot respond to concurrent echo clients until the current client exits. Modify the solution to Exercise 3.18 so that the echo server services each client in a separate request
Answer: Please refer to the supporting Web site for source code solution.

4.19 A naming service such as DNS (for domain name system) can be used to resolve IP names to IP addresses. For example, when someone accesses the host `www.westminstercollege.edu`, a naming service is used to determine the IP address that is mapped to the IP name `www.westminstercollege.edu`. This assignment consists of writing a multithreaded naming service in Java using sockets (see Section 3.6.1.)

The `java.net` API provides the following mechanism for resolving IP names:

```
InetAddress hostAddress =
 InetAddress.getByName("www.westminstercollege.edu");

String IPaddress = hostAddress.getHostAddress();
```

where `getByName()` throws an `UnknownHostException` if it is unable to resolve the host name.

The Server

The server will listen to port 6052 waiting for client connections. When a client connection is made, the server will service the connection in a separate thread and will resume listening for additional client connections. Once a client makes a connection to the server, the client will write the IP name it wishes the server to resolve—such as `www.westminstercollege.edu`—to the socket. The server thread will read this IP name from the socket and either resolve its IP address or, if it cannot locate the host address, catch an `UnknownHostException`. The server will write the IP address back to the client or, in the case of an `UnknownHostException`, will write the message "`Unable to resolve host <host name>`." Once the server has written to the client, it will close its socket connection.

The Client

Initially, write just the server application and connect to it via telnet. For example, assuming the server is running on the localhost, a telnet session would appear as follows. (Client responses appear in blue.)

```
telnet localhost 6052
Connected to localhost.
Escape character is '^]'.
www.westminstercollege.edu
146.86.1.17
Connection closed by foreign host.
```

By initially having telnet act as a client, you can more accurately debug any problems you may have with your server. Once you are convinced your server is working properly, you can write a client application. The client will be passed the IP name that is to be resolved as a parameter. The client will open a socket connection to the server and then write the IP name that is to be resolved. It will then read the response sent back by the server. As an example, if the client is named `NSClient`, it is invoked as follows:

```
java NSClient www.westminstercollege.edu
```

and the server will respond with the corresponding IP address or "unknown host" message. Once the client has output the IP address, it will close its socket connection.

Answer: NO ANSWER

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce the basic scheduling concepts and discuss in great length CPU scheduling. FCFS, SJF, Round-Robin, Priority, and the other scheduling algorithms should be familiar to the students. This is their first exposure to the idea of resource allocation and scheduling, so it is important that they understand how it is done. Gantt charts, simulations, and play acting are valuable ways to get the ideas across. Show how the ideas are used in other situations (like waiting in line at a post office, a waiter time sharing between customers, even classes being an interleaved round-robin scheduling of professors).

A simple project is to write several different CPU schedulers and compare their performance by simulation. The source of CPU and I/O bursts may be generated by random number generators or by a trace tape. The instructor can make up the trace tape in advance to provide the same data for all students. The file that I used was a set of jobs, each job being a variable number of alternating CPU and I/O bursts. The first line of a job was the word JOB and the job number. An alternating sequence of CPU n and I/O n lines followed, each specifying a burst time. The job was terminated by an END line with the job number again. Compare the time to process a set of jobs using FCFS, Shortest-Burst-Time, and round-robin scheduling. Round-robin is more difficult, since it requires putting unfinished requests back in the ready queue.

Exercises

5.9 Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

Answer: I/O-bound programs have the property of performing only a small amount of computation before performing I/O. Such programs typically do not use up their entire CPU quantum. CPU-bound programs, on the other hand, use their entire quantum without performing any blocking I/O operations. Consequently, one could make better use of the

computer's resouces by giving higher priority to I/O-bound programs and allow them to execute ahead of the CPU-bound programs.

5.10 Discuss how the following pairs of scheduling criteria conflict in certain settings.

a. CPU utilization and response time

b. Average turnaround time and maximum waiting time

c. I/O device utilization and CPU utilization

Answer:

a. CPU utilization and response time: CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could, however, result in increasing the response time for processes.

b. Average turnaround time and maximum waiting time: Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could, however, starve long-running tasks and thereby increase their waiting time.

c. I/O device utilization and CPU utilization: CPU utilization is maximized by running long-running CPU-bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O-bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

5.11 Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

a. $\alpha = 0$ and $\tau_0 = 100$ milliseconds

b. $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds

Answer: When $\alpha = 0$ and $\tau_0 = 100$ milliseconds, the formula always makes a prediction of 100 milliseconds for the next CPU burst. When $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memoryless, and simply predicts the length of the previous burst for the next quantum of CPU execution.

5.12 Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.

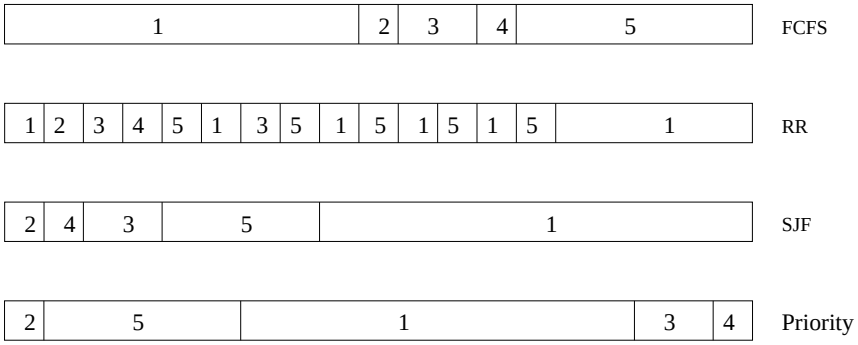
b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of the scheduling algorithms in part a?

d. Which of the schedules in part a results in the minimal average waiting time (over all processes)?

Answer:

a. The four Gantt charts are



b. Turnaround time

	FCFS	RR	SJF	Priority
P_1	10	19	19	16
P_2	11	2	1	1
P_3	13	7	4	18
P_4	14	4	2	19
P_5	19	14	9	6

c. Waiting time (turnaround time minus burst time)

	FCFS	RR	SJF	Priority
P_1	0	9	9	6
P_2	10	1	0	0
P_3	11	5	2	16
P_4	13	3	1	18
P_5	14	9	4	1

d. Shortest Job First

5.13 Which of the following scheduling algorithms could result in starvation?

a. First-come, first-served

b. Shortest job first

c. Round robin

d. Priority

Answer: Shortest job first and priority-based scheduling algorithms could result in starvation.

5.14 Consider a variant of the RR scheduling algorithm where the entries in the ready queue are pointers to the PCBs.

a. What would be the effect of putting two pointers to the same process in the ready queue?

b. What would be the major advantages and disadvantages of this scheme?

c. How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

Answer:

a. In effect, that process will have increased its priority since by getting time more often it is receiving preferential treatment.

b. The advantage is that more important jobs could be given more time, in other words, higher priority in treatment. The consequence, of course, is that shorter jobs will suffer.

c. Allot a longer amount of time to processes deserving higher priority. In other words, have two or more quantums possible in the Round-Robin scheme.

5.15 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context switching overhead is 0.1 millisecond and that all processes are long-running tasks. What is the CPU utilization for a round-robin scheduler when:

a. The time quantum is 1 millisecond

b. The time quantum is 10 milliseconds

Answer:

a. The time quantum is 1 millisecond: Irrespective of which process is scheduled, the scheduler incurs a 0.1 millisecond context-switching cost for every context-switch. This results in a CPU utilization of 1/1.1 * 100 = 91%.

b. The time quantum is 10 milliseconds: The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10*1.1 + 10.1 (as each I/O-bound task executes for 1 millisecond and then incur the context switch task, whereas the

CPU-bound task executes for 10 milliseconds before incurring a context switch). The CPU utilization is therefore 20/21.1 * 100 = 94%.

5.16 Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?

Answer: The program could maximize the CPU time allocated to it by not fully utilizing its time quantums. It could use a large fraction of its assigned quantum, but relinquish the CPU before the end of the quantum, thereby increasing the priority associated with the process.

5.17 Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α; when it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

a. What is the algorithm that results from $\beta > \alpha > 0$?

b. What is the algorithm that results from $\alpha < \beta < 0$?

Answer:

a. FCFS

b. LIFO

5.18 Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:

a. FCFS

b. RR

c. Multilevel feedback queues

Answer:

a. FCFS—discriminates against short jobs since any short jobs arriving after long jobs will have a longer waiting time.

b. RR—treats all jobs equally (giving them equal bursts of CPU time) so short jobs will be able to leave the system faster since they will finish first.

c. Multilevel feedback queues work similar to the RR algorithm—they discriminate favorably toward short jobs.

5.19 Using the Windows XP scheduling algorithm, what is the numeric priority of a thread for the following scenarios?

a. A thread in the REALTIME_PRIORITY_CLASS with a relative priority of HIGHEST.

b. A thread in the NORMAL_PRIORITY_CLASS with a relative priority of NORMAL.

c. A thread in the HIGH_PRIORITY_CLASS with a relative priority of ABOVE_NORMAL.

Answer:

a. 26

b. 8

c. 14

5.20 Consider the scheduling algorithm in the Solaris operating system for time-sharing threads:

a. What is the time quantum (in milliseconds) for a thread with priority 10? With priority 55?

b. Assume a thread with priority 35 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?

c. Assume a thread with priority 35 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

Answer:

a. 160 and 40

b. 35

c. 54

5.21 The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: The higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (Recent CPU usage / 2) + base

where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P_1 is 40, process P_2 is 18, and process P_3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Answer: The priorities assigned to the processes are 80, 69, and 65 respectively. The scheduler lowers the relative priority of CPU-bound processes.

CHAPTER 6

Process Synchronization

Chapter 6 is concerned with the topic of process synchronization among concurrently executing processes. Concurrency is generally very hard for students to deal with correctly, and so we have tried to introduce it and its problems with the classic process coordination problems: mutual exclusion, bounded-buffer, readers/writers, and so on. An understanding of these problems and their solutions is part of current operating-system theory and development.

We first use semaphores and monitors to introduce synchronization techniques. Next, Java synchronization is introduced to further demonstrate a language-based synchronization technique. We conclude with a discussion of how contemporary operating systems provide features for process synchronization and thread safety.

Exercises

6.8 Race conditions are possible in many computer systems. Consider a banking system with the following two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from a bank account. Assume a shared bank account exists between a husband and wife and concurrently the husband calls the `withdraw()` function and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Answer: Assume the balance in the account is 250.00 and the husband calls `withdraw(50)` and the wife calls `deposit(100)`. Obviously the correct value should be 300.00 Since these two transactions will be serialized, the local value of balance for the husband becomes 200.00, but before he can commit the transaction, the `deposit(100)` operation takes place and updates the shared value of balance to 300.00 We then switch back to the husband and the value of the shared balance is set to 200.00 - obviously an incorrect value.

6.9 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 6.25; the other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

Answer: This algorithm satisfies the three conditions of mutual exclusion. (1) Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn. (2) Progress is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting turn to the other process upon exiting. (3) Bounded waiting is preserved through the use of the TTturn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

6.10 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

All the elements of flag are initially idle; the initial value of turn is immaterial (between 0 and n-1). The structure of process P_i is shown in Figure 6.26. Prove that the algorithm satisfies all three requirements for the critical-section problem.

Answer: This algorithm satisfies the three conditions. Before we show that the three conditions are satisfied, we give a brief explanation of what the algorithm does to ensure mutual exclusion. When a process i requires access to critical section, it first sets its flag variable to want_in to indicate its desire. It then performs the following steps: (1) It ensures that all processes whose index lies between turn and i are idle. (2) If so, it updates its flag to in_cs and checks whether there is already some other process that has updated its flag to in_cs. (3) If not and if it is this process's turn to enter the critical section or if the process

indicated by the `turn` variable is idle, it enters the critical section. Given the above description, we can reason about how the algorithm satisfies the requirements in the following manner:

a. Mutual exclusion is ensured: Notice that a process enters the critical section only if the following requirements is satisfied: no other process has its `flag` variable set to in_cs. Since the process sets its own `flag` variable set to in_cs before checking the status of other processes, we are guaranteed that no two processes will enter the critical section simultaneously.

b. Progress requirement is satisfied: Consider the situation where multiple processes simultaneously set their `flag` variables to in_cs and then check whether there is any other process has the `flag` variable set to in_cs. When this happens, all processes realize that there are competing processes, enter the next iteration of the outer *while*(1) loop and reset their `flag` variables to want_in. Now the only process that will set its `turn` variable to in_cs is the process whose index is closest to `turn`. It is however possible that new processes whose index values are even closer to `turn` might decide to enter the critical section at this point and therefore might be able to simultaneously set its `flag` to in_cs. These processes would then realize there are competing processes and might restart the process of entering the critical section. However, at each iteration, the index values of processes that set their `flag` variables to in_cs become closer to `turn` and eventually we reach the following condition: only one process (say `k`) sets its `flag` to in_cs and no other process whose index lies between `turn` and k has set its `flag` to in_cs. This process then gets to enter the critical section.

c. Bounded-waiting requirement is met: The bounded waiting requirement is satisfied by the fact that when a process k desires to enter the critical section, its `flag` is no longer set to idle. Therefore, any process whose index does not lie between `turn` and k cannot enter the critical section. In the meantime, all processes whose index falls between `turn` and k and desire to enter the critical section would indeed enter the critical section (due to the fact that the system always makes progress) and the `turn` value monotonically becomes closer to k. Eventually, either `turn` becomes k or there are no processes whose index values lie between `turn` and k, and therefore process k gets to enter the critical section.

6.11 What is the meaning of the term *busy waiting*? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Answer: *Busy waiting* means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

6.12 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.
Answer: Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

6.13 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.
Answer: If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.

6.14 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
Answer: Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

6.15 Describe two kernel data structures where race conditions are possible. Be sure to include a dscription describing how a race can occur.
Answer: There are many answers to this question. Some kernel data structures include a process id (pid) management system, kernel process table, and scheduling queues. With a pid management system, it is possible two processes may be created at the same time and there is a race condition assigning each process a unique pid. The same type of race condition can occur in the kernel process table: two processes are created at the same time and there is a race assigning them a location in the kernel process table. With scheduling queues, it is possible one process has been waiting for IO which is now available. Another process is being context-switched out. These two processes are being moved to the Runnable queue at the same time. Hence there is a race condition in the Runnable queue.

6.16 Describe how the `Swap()` instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.
Answer:

```
do {waiting[i] = TRUE; key = TRUE;
    while (waiting[i] && key) key = Swap(&lock, &key);

    waiting[i] = FALSE;

    /* critical section */

    j = (i+1) % n; while ((j != i) && !waiting[j])
    j = (j+1) % n;
    if (j == i) lock = FALSE; else waiting[j] = FALSE;

    n/* remainder section */
    while (TRUE);
}
```

6.17 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.
Answer: A semaphore is initialized to the number of allowable open socket connections. When a connection is accepted, the `acquire()` method is called; when a connection is released, the `release()` method is called. If the system reaches the number of allowable socket connections, subsequent calls to `acquire()` will block until an existing connection is terminated and the release method is invoked.

6.18 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.
Answer: A wait operation atomically decrements the value associated with a semaphore. If two wait operations are executed on a semaphore when its value is 1, if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value, thereby violating mutual exclusion.

6.19 Windows Vista provides a new lightweight synchronization tool called **slim reader–writer** locks. Whereas most implementations of reader–writer locks favor either readers or writers or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers or writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.
Answer: Simplicity. IF RW locks provide fairness or favor readers or writers, there is more overhead to the lock. By providing such a simple synchronization mechanism, access to the lock is fast. Usage of this lock may be most appropriate for situations where reader–locks are needed, but quickly acquiring and releasing the lock is similarly important.

6.20 Show how to implement the `wait()` and `signal()` semaphore operations in multiprocessor environments using the `TestAndSet()` instruction. The solution should exhibit minimal busy waiting.

Answer: Here is the pseudocode for implementing the operations:

```
int guard = 0;
int semaphore_value = 0;

wait()
{
    while (TestAndSet(&guard) == 1);
    if (semaphore_value == 0) {
       atomically add process to a queue of processes
       waiting for the semaphore and set guard to 0;
    }else {
       semaphore_value--;
       guard = 0;
    }
}


signal()
{
    while (TestAndSet(&guard) == 1);
    if (semaphore_value == 0 &&
            there is a process on the wait queue)
        wake up the first process in the queue
        of waiting processes
    else
       semaphore_value++;
    guard = 0;
}
```

6.21 Exercise 4.17 requires the parent thread to wait for the child thread to finish its execution before printing out the computed values. If we let the parent thread access the Fibonacci numbers as soon as they have been computed by the child thread — rather than waiting for the child thread to terminate — Explain what changes would be necessary to the solution for this exercise? Implement your modified solution.

Answer: A counting sempahore or condition variable works fine. The sempahore would be initialized to zero, and the parent would call the wait() function. When completed, the child would invoke signal(), thereby notifying the parent. If a condition variable is used, the parent thread will invoke wait() and the child will call signal() when completed. In both instances, the idea is that the parent thread waits for the child for notification that its data is available.

6.22 Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement the same types of synchronization problems.

Answer: A semaphore can be implemented using the following monitor code:

```
monitor semaphore {
    int value = 0;
    condition c;

    semaphore_increment() {
       value++;
       c.signal();
    }

    semaphore_decrement() {
       while (value == 0)
       c.wait();
       value--;
    }
}
```

A monitor could be implemented using a semaphore in the following manner. Each condition variable is represented by a queue of threads waiting for the condition. Each thread has a semaphore associated with its queue entry. When a thread performs a wait operation, it creates a new semaphore (initialized to zero), appends the semaphore to the queue associated with the condition variable, and performs a blocking semaphore decrement operation on the newly created semaphore. When a thread performs a signal on a condition variable, the first process in the queue is awakened by performing an increment on the corresponding semaphore.

6.23 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.
Answer:

```
monitor bounded_buffer {
    int items[MAX_ITEMS];
    int numItems = 0;
    condition full, empty;

    void produce(int v) {
       while (numItems == MAX_ITEMS) full.wait();
       items[numItems++] = v;
       empty.signal();
    }

    int consume() {
       int retVal;
       while (numItems == 0) empty.wait();
       retVal = items[--numItems];
       full.signal();
       return retVal;
    }
}
```

6.24 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.23 mainly suitable for small portions.

a. Explain why this is true.

b. Design a new scheme that is suitable for larger portions.

Answer: The solution to the bounded buffer problem given above copies the produced value into the monitor's local buffer and copies it back from the monitor's local buffer to the consumer. These copy operations could be expensive if one were using large extents of memory for each buffer region. The increased cost of copy operation means that the monitor is held for a longer period of time while a process is in the produce or consume operation. This decreases the overall throughput of the system. This problem could be alleviated by storing pointers to buffer regions within the monitor instead of storing the buffer regions themselves. Consequently, one could modify the code given above to simply copy the pointer to the buffer region into and out of the monitor's state. This operation should be relatively inexpensive and therefore the period of time that the monitor is being held will be much shorter, thereby increasing the throughput of the monitor.

6.25 Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.
Answer: Throughput in the readers-writers problem is increased by favoring multiple readers as opposed to allowing a single writer to exclusively access the shared values. On the other hand, favoring readers could result in starvation for writers. The starvation in the readers-writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wake up the process that has been waiting for the longest duration. When a reader arrives and notices that another reader is accessing the database, then it would enter the critical section only if there are no waiting writers. These restrictions would guarantee fairness.

6.26 How does the `signal()` operation associated with monitors differ from the corresponding operation defined for semaphores?
Answer: The `signal()` operation associated with monitors is not persistent in the following sense: if a signal is performed and if there are no waiting threads, then the signal is simply ignored and the system does not remember that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks. In semaphores, on the other hand, every signal results in a corresponding increment of the semaphore value even if there are no waiting threads. A future wait operation would immediately succeed because of the earlier increment.

6.27 Suppose the `signal()` statement can appear only as the last statement in a monitor procedure. Suggest how the implementation described in Section 6.7 can be simplified.
Answer: If the signal operation were the last statement, then the lock could be transferred from the signalling process to the process that is the

recipient of the signal. Otherwise, the signalling process would have to explicitly release the lock and the recipient of the signal would have to compete with all other processes to obtain the lock to make progress.

6.28 Consider a system consisting of processes P_1, P_2, ..., P_n, each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.

Answer: Here is the pseudocode:

```
monitor printers {
    int num_avail = 3;
    int waiting_processes[MAX_PROCS];
    int num_waiting;
    condition c;

    void request_printer(int proc_number) {
      if (num_avail > 0) {
      num_avail--;
      return;
      }
      waiting_processes[num_waiting] = proc_number;
      num_waiting++;
      sort(waiting_processes);
      while (num_avail == 0 &&
             waiting_processes[0] != proc_number)
      c.wait();
      waiting_processes[0] =
             waiting_processes[num_waiting-1];
      num_waiting--;
      sort(waiting_processes);
      num_avail--;
    }

    void release_printer() {
      num_avail++;
      c.broadcast();
    }
}
```

6.29 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.

Answer: The pseudocode is as follows:

```
monitor file_access {
    int curr_sum = 0;
    int n;
    condition c;

    void access_file(int my_num) {
       while (curr_sum + my_num >= n)
       c.wait();
       curr_sum += my_num;
    }

    void finish_access(int my_num) {
       curr_sum -= my_num;
       c.broadcast();
    }
}
```

6.30 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with the two different ways in which signaling can be performed?
Answer: The solution to the previous exercise is correct under both situations. However, it could suffer from the problem that a process might be awakened only to find that it is still not possible for it to make forward progress either because there was not sufficient slack to begin with when a process was awakened or if an intervening process gets control, obtains the monitor and starts accessing the file. Also, note that the broadcast operation wakes up all of the waiting processes. If the signal also transfers control and the monitor from the current thread to the target, then one could check whether the target would indeed be able to make forward progress and perform the signal only if it it were possible. Then the "while" loop for the waiting thread could be replaced by an "if" condition since it is guaranteed that the condition will be satisfied when the process is woken up.

6.31 Suppose we replace the `wait()` and `signal()` operations of monitors with a single construct `await(B)`, where `B` is a general Boolean expression that causes the process executing it to wait until `B` becomes `true`.

a. Write a monitor using this scheme to implement the readers–writers problem.

b. Explain why, in general, this construct cannot be implemented efficiently.

c. What restrictions need to be put on the `await` statement so that it can be implemented efficiently? (Hint: Restrict the generality of `B`; see Kessels [1977].)

Answer:

a. The readers–writers problem could be modified with the following more generate `await` statements:
A reader can perform "await(active_writers == 0 && waiting_writers == 0)" to check that there are no active writers and there are no waiting writers before it enters the critical section. The writer can perform a "await(active_writers == 0 && active_readers == 0)" check to ensure mutually exclusive access.

b. The system would have to check which one of the waiting threads have to be awakened by checking which one of their waiting conditions are satisfied after a signal. This requires considerable complexity and might require some interaction with the compiler to evaluate the conditions at different points in time. One could restrict the Boolean condition to be a disjunction of conjunctions with each component being a simple check (equality or inequality with respect to a static value) on a program variable. In that case, the Boolean condition could be communicated to the run-time system, which could perform the check every time it needs to determine which thread to be awakened.

6.32 Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a procedure *tick* in your monitor at regular intervals.
Answer: Here is a pseudocode for implementing this:

```
monitor alarm {
    condition c;

    void delay(int ticks) {
       int begin_time = read_clock();
       while (read_clock() < begin_time + ticks)
       c.wait();
    }

    void tick() {
       c.broadcast();
    }
}
```

6.33 Why do Solaris, Linux, and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems and not on single-processor systems?
Answer: Solaris, Linux, and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems. Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock could be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program

condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

6.34 In log-based systems that provide support for transactions, updates to data items cannot be performed before the corresponding entries are logged. Why is this restriction necessary?
Answer: If the transaction needs to be aborted, then the values of the updated data values need to be rolled back to the old values. This requires the old values of the data entries to be logged before the updates are performed.

6.35 Show that the two-phase locking protocol ensures conflict serializability.
Answer: A schedule refers to the execution sequence of the operations for one or more transactions. A serial schedule is the situation where each transaction of a schedule is performed atomically. If a schedule consists of two different transactions where consecutive operations from the different transactions access the same data and at least one of the operations is a write, then we have what is known as a *conflict*. If a schedule can be transformed into a serial schedule by a series of swaps on nonconflicting operations, we say that such a schedule is conflict serializable.
The two-phase locking protocol ensures conflict serializabilty because exclusive locks (which are used for write operations) must be acquired serially, without releasing any locks during the acquire (growing) phase. Other transactions that wish to acquire the same locks must wait for the first transaction to begin releasing locks. By requiring that all locks must first be acquired before releasing any locks, we are ensuring that potential conflicts are avoided.

6.36 What are the implications of assigning a new timestamp to a transaction that is rolled back? How does the system process transactions that were issued after the rolled-back transaction but that have timestamps smaller than the new timestamp of the rolled-back transaction?
Answer: If the transactions that were issued after the rolled-back transaction had accessed variables that were updated by the rolled-back transaction, then these transactions would have to be rolled back as well. If they have not performed such operations (that is, there is no overlap with the rolled-back transaction in terms of the variables accessed), then these operations are free to commit when appropriate.

6.37 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and —once finished—will return them. As an example, many commercial software packages provide a given number of **licenses**, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will be granted only when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the `decrease_count()` function:

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
  if (available_resources < count)
    return -1;
  else {
    available_resources -= count;

    return 0;
  }
}
```

When a process wants to return a number of resources, it calls the `decrease_count()` function:

```
/* increase available_resources by count */
int increase_count(int count) {
  available_resources += count;

  return 0;
}
```

The preceding program segment produces a race condition. Do the following:

a. Identify the data involved in the race condition.

b. Identify the location (or locations) in the code where the race condition occurs.

c. Using a semaphore, fix the race condition.

Answer:

- Identify the data involved in the race condition: The variable available_resources.
- Identify the location (or locations) in the code where the race condition occurs: The code that decrements available_resources and the code that increments available_resources are the statements that could be involved in race conditions.

- Using a semaphore, fix the race condition: Use a semaphore to represent the available_resources variable and replace increment and decrement operations by semaphore increment and semaphore decrement operations.

6.38 The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and -1 otherwise. This leads to awkward programming for a process that wishes obtain a number of resources:

```
while (decrease_count(count) == -1)
   ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

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

Answer:

```
monitor resources {
    int available_resources;
    condition resources_avail;

    int decrease_count(int count) {
      while (available_resources < count)
      resources_avail.wait();
      available_resources -= count; }

    int increase_count(int count) {
      available_resources += count;
      resources_avail.signal();
}
```