INSTRUCTOR'S MANUAL
TO ACCOMPANY

Operating System Concepts

Ninth Edition

Abraham Silberschatz
Yale University

Peter Baer Galvin
Pluribus Networks

Greg Gagne
Westminster College

Contents

Preface

This volume is an instructor's manual for the Ninth Edition of *Operating System Concepts* by Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. It consists of answers to the exercises in the parent text.

Although we have tried to produce an instructor's manual that will aid all of the users of our book as much as possible, there can always be improvements (improved answers, additional questions, sample test questions, programming projects, alternative orders of presentation of the material, additional references, and so on). We invite you to help us in improving this manual. If you have better solutions to the exercises or other items that would be of use with *Operating-System Concepts*, we invite you to send them to us for consideration in later editions of this manual. All contributions will, of course, be properly credited to their contributor. Email should be addressed to os-book-authors@cs.yale.edu.

A. S.
P. B. G.
G. G.

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

1.12 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.

1.13 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

1.14 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.

1.15 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.

1.16 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.

1.17 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.

1.18 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.

1.19 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.

1.20 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.

1.21 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.

1.22 Many SMP systems have different levels of caches; one level is local to each processing core, and another level is shared among all processing cores. Why are caching systems designed this way?
Answer:
The different levels are based on access speed as as well as size. In general, the closer the cache is to the CPU, the faster the access. However, faster caches are typically more costly. Therefore, smaller and faster caches are placed local to each CPU, and shared caches that are larger, yet slower, are shared among several different processors.

1.23 Consider an SMP system similar to the one shown in Figure 1.6. Illustrate with an example how data residing in memory could in fact have a different value 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.

1.24 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.

1.25 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.

1.26 Which network configuration—LAN or WAN—would best suit the following environments?

a. A campus student union

b. Several campus locations across a statewide university system

c. A neighborhood

Answer:

a. LAN

b. WAN

c. LAN or WAN

1.27 Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs.
Answer:
The greatest challenges in designing mobile operating systems include:

- Less storage capacity means the operating system must manage memory carefully.
- The operating system must also manage power consumption carefully.

- Less processing power plus fewer processors mean the operating system must carefully apportion processors to applications.

1.28 What are some advantages of peer-to-peer systems over client-server systems?
Answer:
Peer-to-peer is useful because services are distributed across a collection of peers, rather than having a single, centralized server. Peer-to-peer provides fault tolerance and redundancy. Also, because peers constantly migrate, they can provide a level of security over a server that always exists at a known location on the Internet. Peer-to-peer systems can also potentially provide higher network bandwidth because you can collectively use all the bandwidth of peers, rather than the single bandwidth that is available to a single server.

1.29 Describe some distributed applications that would be appropriate for a peer-to-peer system.
Answer:
Essentially anything that provides content,in addition to existing services such as file services, distributed directory services such as domain name services, and distributed e-mail services.

1.30 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:

The two models of interprocess communication are message-passing model and the shared-memory model. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. However, this method compromises on protection and synchronization between the processes sharing memory.

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 What are the advantages of using loadable kernel modules?
Answer:
It is difficult to predict what features an operating system will need when it is being designed. The advantage of using loadable kernel modules is that functionality can be added to and removed from the kernel while it is running. There is no need to either recompile or reboot the kernel.

2.23 How are iOS and Android similar? How are they different?
Answer:
Similarities

- Both are based on existing kernels (Linux and Mac OS X).
- Both have architecture that uses software stacks.
- Both provide frameworks for developers.

Differences

- iOS is closed-source, and Android is open-source.
- iOS applications are developed in Objective-C, Android in Java.
- Android uses a virtual machine, and iOS executes code natively.

2.24 Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.
Answer:
It is because the standard API and virtual machine are designed for desktop and server systems, not mobile devices. Google developed a separate API and virtual machine for mobile devices.

2.25 The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and optimization of system performance.
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

3.8 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.

3.9 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.

3.10 Construct a process tree similar to Figure 3.8. To obtain process information for the UNIX or Linux system, use the command `ps -ael`. Use the command `man ps` to get more information about the `ps` command. The task manager on Windows systems does not provide the parent process id,yet the **process monitor** tool available from `technet.microsoft.com` provides a process tree tool.
Answer:
Answer: Results will vary widely.

3.11 Explain the role of the `init` process on UNIX and Linux systems in regards to process termination.
Answer:
When a process is terminated, it briefly moves to the zombie state and remains in that state until the parent invokes a call to `wait()`. When this occurs, the process id as well as entry in the process table are both released. However,if a parent does not invoke `wait()`, the child process remains a zombie as long as the parent remains alive. Once the parent process terminates, the `init`process becomes the new parent of the zombie. Periodically, the `init` process calls `wait()` which ultimately releases the pid and entry in the process table of the zombie process.

3.12 Including the initial parent process, how many processes are created by the program shown in Figure Figure 3.32?
Answer:
8 processes are created. The program online includes printf() statements to better understand how many processes have been created.

3.13 Explain the circumstances when the line of code marked `printf("LINE J")` in Figure 3.33 is reached.
Answer:
The call to `exec()` replaces the address space of the process with the program specified as the parameter to `exec()`. If the call to `exec()` succeeds, the new program is now running and control from the call to `exec()` never returns. In this scenario, the line `printf("Line J");` would never be performed. However, if an error occurs in the call to `exec()`, the function returns control and therefor the line `printf("Line J");` would be performed.

3.14 Using the program in Figure Figure 3.34, identify the values of `pid` at lines `A`, `B`, `C`, and `D`. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)
Answer:
Answer: A = 0, B = 2603, C = 2603, D = 2600

3.15 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.

3.16 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

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 semantics, 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 account information such as name, address, etc. does not require these semantics.

3.17 Using the program shown in Figure 3.35, explain what the output will be at lines `X` and `Y`.

Answer:

Because the child is a copy of the parent, any changes the child makes will occur in its copy of the data and won't be reflected in the parent. As a result, the values output by the child at line X are 0, -1, -4, -9, -16. The values output by the parent at line Y are 0, 1, 2, 3, 4

3.18 What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.

a. Synchronous and asynchronous communication

b. Automatic and explicit buffering

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

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

4.6 Provide two programming examples in which multithreading does **not** provide better performance than a single-threaded Solution.
Answer:

a. 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.

b. 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.

4.7 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.

4.8 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.

4.9 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.
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.

4.10 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new website in a separate process. Would the same benefits have been achieved if instead Chrome had been designed to open each new website in a separate thread? Explain.
Answer:
No. The primary reason for opening each website in a separate process is that if a web application in one website crashes, only that renderer process is affected, and the browser process, as well as other renderer processes, are unaffected. Because multiple threads all belong to the same process, any thread that crashes would affect the entire process.

4.11 Is it possible to have concurrency but not parallelism? Explain.
Answer:
Yes. Concurrency means that more than one process or thread is progressing at the same time. However, it does not imply that the processes are running simultaneously. The scheduling of tasks allows for concurrency, but parallelism is supported only on systems with more than one processing core.

4.12 Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.
Answer:
Two processing cores = 1.43 speedup; four processing cores = 1.82 speedup.

4.13 Determine if the following problems exhibit task or data parallelism:

- The multithreaded statistical program described in Exercise 4.21
- The multithreaded Sudoku validator described in Project 1 in this chapter
- The multithreaded sorting program described in Project 2 in this chapter
- The multithreaded web server described in Section 4.1

Answer:

- Task parallelism. Each thread is performing a different task on the same set of data.
- Task parallelism. Each thread is performing a different task on the same data.
- Data parallelism. Each thread is performing the same task on different subsets of data.
- Task parallelism. Likely running the same code, but on entirely different data.

4.14 A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between startup and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

Answer:

- It only makes sense to create as many threads as there are blocking system calls, as the threads will be spent blocking. Creating additional threads provides no benefit. Thus, it makes sense to create a single thread for input and a single thread for output.
- Four. There should be as many threads as there are processing cores. Fewer would be a waste of processing resources, and any number > 4 would be unable to run.

4.15 Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
   fork();
   thread_create( . . .);
}
fork();
```

a. How many unique processes are created?

b. How many unique threads are created?

Answer:
There are six processes and two threads.

4.16 As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. However, 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 requiredto identify which threads correspond to which process and perform therelevant accounting tasks.

4.17 The program shown in Figure 4.16 uses the Pthreads API. What would be the output from the program at `LINE C` and `LINE P`?
Answer:
Output at `LINE C` is 5. Output at `LINE P` is 0.

4.18 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 more 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.

4.19 Pthreads provides an API for managing thread cancellation. The `pthread_setcancelstate()` function is used to set the cancellation state. Its prototype appears as follows:

```
pthread_setcancelstate(int state, int *oldstate)
```

The two possible values for the state are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

Using the code segment shown in Figure 4.17, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

Answer:
Three examples:

a. An update to a file

b. A situation in which two write operations must both complete if either completes

c. Essentially any operation that we want to run to completion

CHAPTER 5

Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads, discussed in Chapter 4. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Exercises

5.7 Race conditions are possible in many computer systems. Consider a banking system with two methods: `deposit(amount)` and `withdraw(amount)`. These two methods are passed the `amount` that is to be deposited or withdrawn from a bank account. Assume that a husband and wife share a bank account and that concurrently the husband calls the `withdraw()` method 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.

5.8 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 5.21; 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.

5.9 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 5.22. 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.

5.10 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.

5.11 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.

5.12 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.
Answer:
Because acquiring a semaphore may put the process to sleep while it is waiting for the semaphore to become available. Spinlocks are to only be held for short durations and a process that is sleeping may hold the spinlock for too long a period.

5.13 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition 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.

5.14 Describe how the `compare_and_swap()` instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.
Answer:
Please see Figure 5.1

5.15 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
     int available;
} lock;
```

```
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);
}
```

Figure 5.1 Program for Exercise 5.14.

where (`available == 0`) indicates the lock is available; a value of 1 indicates the lock is unavailable. Using this `struct`, illustrate how the following functions may be implemented using the `test_and_set()` and `compare_and_swap()` instructions.

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.
Answer:
Please see Figure 5.2

5.16 The implementation of mutex locks provided in Section 5.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.
Answer:
This would be very similar to the changes made in the description of the semaphore. Associated with each mutex lock would be a queue of waiting processes. When a process determines the lock is unavailable, they are placed into the queue. When a process releases the lock, it removes and awakens the first process from the list of waiting processes.

5.17 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- The thread may be put to sleep while holding the lock.

```
// initialization
mutex->available = 0;

// acquire using compare_and_swap()
void acquire(lock *mutex) {
   while (compare_and_swap(&mutex->available, 0, 1) != 0)
     ;

   return;
}


// acquire using test_and_set()
void acquire(lock *mutex) {
   while (test_and_set(&mutex->available) != 0)
     ;

   return;
}


void release(lock *mutex) {
   mutex->available = 0;

   return;
}
```

Figure 5.2 Program for Exercise 5.15.

Answer:

- Spinlock
- Mutex lock
- Mutex lock

5.18 Assume a context switch takes T time. Ssuggest an upper bound (in terms of T) for holding a spin lock and that if the spin lock is held for any longer duration, a mutex lock (where waiting threads are put to sleep) is a better alternative.

Answer:
The spinlock should be held for $< 2xT$. Any longer than this duration it would be faster to put the thread to sleep (requiring one context switch) and then subsequently awaken it (requiring a second context switch.)

5.19 A multithreaded web server wishes to keep track of the number of requests it services (known as **hits**.) Consider the following two strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

Answer:

The use of locks is overkill in this situation. Locking generally requires a system call and possibly putting a process to sleep (and thus requiring a context switch) if the lock is unavailable. (Awakening the process will similarly require another subsequent context switch.) On the other hand, the atomic integer provides an atomic update of the `hits` variable and ensures no race condition on `hits`. This can be accomplished with no kernel intervention and therefore the second approach is more efficient.

5.20 Consider the code example for allocating and releasing processes shown in Figure 5.23.

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0
```

to prevent the race condition(s)?

Answer:

a. There is a race condition on the variable `number_of_processes`.

b. A call to `acquire()` must be placed upon entering each function and a call to `release()` immediately before exiting each function.

c. No, it would not help. The reason is because the race occurs in the `allocate_process()` function where `number_of_processes` is first tested in the `if` statement, yet is updated afterwards, based upon the value of the test. it is possible that `number_of_processes = 254` at the time of the test, yet because of the race condition, is set to 255 by another thread before it is incremented yet again.

5.21 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.

5.22 Windows Vista provides a new lightweight synchronization tool called a **slim reader–writer lock**. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers and do not order waiting threads 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.

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

Answer:

Here is the pseudocode for implementing the operations:
Please see Figure 5.3

5.24 Exercise 4.26 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—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.

```
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;
}
```

Figure 5.3 Program for Exercise 5.23.

5.25 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:
Please see Figure 5.4
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.

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

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

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

Figure 5.4 Program for Exercise 5.25.

5.26 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

Answer:

Please see Figure 5.5

5.27 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 5.26 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.

```
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;
    }
}
```

Figure 5.5 Program for Exercise 5.26.

5.28 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.

5.29 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.

```
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();
    }
}
```

Figure 5.6 Program for Exercise 5.31.

5.30 Suppose the `signal()` statement can appear only as the last statement in a monitor procedure. Suggest how the implementation described in Section 5.8 can be simplified in this situation.
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.

5.31 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:
The pseudocode is presented in Figure 5.6

5.32 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several

```
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();
    }
}
```

Figure 5.7 Program for Exercise 5.32.

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 presented in Figure 5.7.

5.33 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways of performing signaling?
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.

5.34 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.

c. Please see Kessels [1977].

5.35 Design an algorithm for a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.

Answer:

A pseudocode for implementing this is presented in Figure 5.8

```
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();
   }
}
```

Figure 5.8 Program for Exercise 5.35.

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

6.10 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.

6.11 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.

6.12 One technique for implementing **lottery scheduling** works by assigning processes lottery tickets, which are used for allocating CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the CPU. The BTV operating system implements lottery scheduling by holding a lottery 50 times each second, with each lottery winner getting 20 milliseconds of CPU time (20 milliseconds $\times$ 50 $=$ 1 second). Describe how the BTV scheduler can ensure that higher-priority threads receive more attention from the CPU than lower-priority threads.
Answer:
By assigning more lottery tickets to higher-priority processes.

6.13 In Chapter 5, we discussed possible race conditions on various kernel data structures. Most scheduling algorithms maintain a **run queue**, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. What are the advantages and disadvantages of each of these approaches?
Answer:
The primary advantage of each processing core having its own run queue is that there is no contention over a single run queue when the scheduler is running concurrently on 2 or more processors. When a scheduling decision must be made for a processing core, the scheduler only need to look no further than its private run queue. A disadvantage of a single run

queue is that it must be protected with locks to prevent a race condition and a processing core may be available to run a thread, yet it must first acquire the lock to retrieve the thread from the single queue. However, load balancing would likely not be an issue with a single run queue, whereas when each processing core has its own run queue, there must be some sort of load balancing between the different run queues.

6.14 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.

6.15 A variation of the round-robin scheduler is the **regressive round-robin scheduler**. This scheduler assigns each process a time quantum and a priority. The initial value of a time quantum is 50 milliseconds. However, every time a process has been allocated the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted. (The time quantum for a process can be increased to a maximum of 100 milliseconds.) When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the same. What type of process (CPU-bound or I/O-bound) does the regressive round-robin scheduler favor? Explain.
Answer:
This scheduler would favor CPU-bound processes as they are rewarded with a longer time quantum as well as priority boost whenever they consume an entire time quantum. This scheduler does not penalize I/O-bound processes as they are likely to block for I/O before consuming their entire time quantum, but their priority remains the same.

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

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

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

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

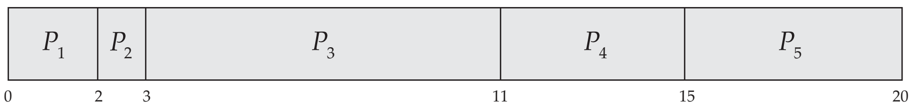
b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

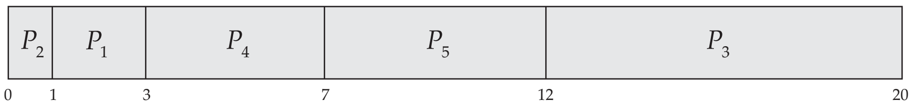
c. What is the waiting time of each process for each of these scheduling algorithms?

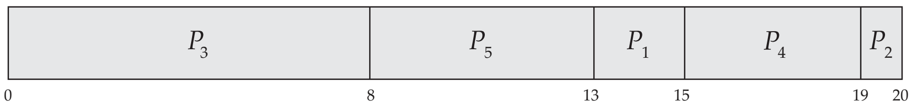
d. Which of the algorithms results in the minimum average waiting time (over all processes)?

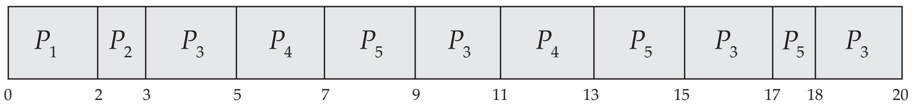
Answer:

a. The four Gantt charts are









b. Turnaround time

	FCFS	SJF	Priority	RR
P_1	2	3	15	2
P_2	3	1	20	3
P_3	11	20	8	20
P_4	15	7	19	13
P_5	20	12	13	18

c. Waiting time (turnaround time minus burst time)

	FCFS	SJF	Priority	RR
P_1	0	1	13	0
P_2	2	0	19	2
P_3	3	12	0	12
P_4	11	3	15	9
P_5	15	7	8	13

d. Shortest Job First

6.17 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an **idle task** (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

d. What is the CPU utilization rate?

Thread	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

Answer:

a. Gantt chart in handwritten notes.

b. p1: 20-0 - 20, p2: 80-25 = 55, p3: 90 - 30 = 60, p4: 75-60 = 15, p5: 120-100 = 20, p6: 115-105 = 10

c. 1 p1: 0, p2: 40, p3: 35, p4: 0, p5: 10, p6: 0

d. 105/120 = 87.5 percent.

6.18 The `nice` command is used to set the nice value of a process on Linux, as well as on other UNIX systems. Explain why some systems may allow any user to assign a process a nice value $>= 0$ yet allow only the root user to assign nice values < 0.

Answer:

Nice values < 0 are assigned a higher relative priority and such systems may not allow non-root processes to assign themselves higher priorities.

6.19 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.

6.20 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 two 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.

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

6.22 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.

6.23 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

6.24 Explain the differences in how much 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.

6.25 Using the Windows scheduling algorithm, determine the numeric priority of each of the following threads

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

6.26 Assuming that no threads belong to the REALTIME_PRIORITY_CLASS and that none may be assigned a TIME_CRITICAL priority, what combination of priority class and priority corresponds to the highest possible relative priority in Windows scheduling?
Answer:
HIGH priority class and HIGHEST priority within that class. (numeric priority of 15)

6.27 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

6.28 Assume that two tasks *A* and *B* are running on a Linux system. The nice values of *A* and *B* are −5 and +5, respectively. Using the CFS scheduler as a guide, describe how the respective values of `vruntime` vary between the two processes given each of the following scenarios:

- Both *A* and *B* are CPU-bound.
- *A* is I/O-bound, and *B* is CPU-bound.
- *A* is CPU-bound, and *B* is I/O-bound.

Answer:

- Since A has a higher priority than B, `vruntime` will move more slowly for A than B. If both A and B are CPU-bound (that is they both use the CPU for as long as it is allocated to them), `vruntime` will generally be smaller for A than B, and hence A will have a greater priority to run over B.
- In this situation, `vruntime` will be much smaller for A than B as (1) `vruntime` will move more slowly for A than B due to priority differences, and (2) A will require less CPU-time as it is I/O-bound.
- This situation is not as clear, and it is possible that B may end up running in favor of A as it will be using the processor less than A and in fact its value of `vruntime` may in fact be less than the value of `vruntime` for B.

6.29 Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

Answer:
The priority inversion problem could be addressed by temporarily changing the priorities of the processes involved. Processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priority reverts to its original value. This solution can be easily implemented within a proportional share scheduler; the shares of the high-priority processes are simply transferred to the lower-priority process for the duration when it is accessing the resources.

6.30 Under what circumstances is rate-monotonic scheduling inferior to earliest-deadline-first scheduling in meeting the deadlines associated with processes?

Answer:
Consider two processes P_1 and P_2 where $p_1 = 50$, $t_1 = 25$ and $p_2 = 75$, $t_2 = 30$. If P_1 were assigned a higher priority than P_2, then the following scheduling events happen under rate-monotonic scheduling. P_1 is scheduled at $t = 0$, P_2 is scheduled at $t = 25$, P_1 is scheduled at $t = 50$, and P_2 is scheduled at $t = 75$. P_2 is not scheduled early enough to meet its deadline. The earliest deadline schedule performs the following scheduling events: P_1 is scheduled at $t = 0$, P_2 is scheduled at $t = 25$, P_1 is scheduled at $t = 55$, and so on. This schedule actually meets the deadlines and therefore earliest-deadline-first scheduling is more effective than the rate-monotonic scheduler.

6.31 Consider two processes, P_1 and P_2, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.

a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart such as the ones in Figure 6.16–Figure 6.19.

b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

Answer:
Consider when P_1 is assigned a higher priority than P_2 with the rate monotonic scheduler. P_1 is scheduled at $t = 0$, P_2 is scheduled at $t = 25$, P_1 is scheduled at $t = 50$, and P_2 is scheduled at $t = 75$. P_2 is not scheduled early enough to meet its deadline. When P_1 is assigned a lower priority than P_2, then P_1 does not meet its deadline since it will not be scheduled in time.

6.32 Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.
Answer:
following tasks: save the currently executing instruction, determine the type of interupt, save the current process state, and then invoke the appropriate interrupt service routine. Dispatch latency is the cost associated with stopping one process and starting another. Both interrupt and dispatch latency needs to be minimized in order to ensure that real-time tasks receive immediate attention. Furthermore, sometimes interrupts are disabled when kernel data structures are being modified, so the interrupt does not get serviced immediately. For hard real-time systems, the time-period for which interrupts are disabled must be bounded in order to guarantee the desired quality of service.

CHAPTER 7

Deadlocks

Deadlock is a problem that can arise only in a system with multiple active asynchronous processes. It is important that the students learn the three basic approaches to deadlock: prevention, avoidance, and detection (although the terms *prevention* and *avoidance* are easy to confuse).

It can be useful to pose a deadlock problem in human terms and ask why human systems never deadlock. Can the students transfer this understanding of human systems to computer systems?

Projects can involve simulation: create a list of jobs consisting of requests and releases of resources (single type or multiple types). Ask the students to allocate the resources to prevent deadlock. This basically involves programming the Banker's Algorithm.

The survey paper by Coffman, Elphick, and Shoshani [1971] is good supplemental reading, but you might also consider having the students go back to the papers by Havender [1968], Habermann [1969], and Holt [1971a]. The last two were published in *CACM* and so should be readily available.

Exercises

7.11 Consider the traffic deadlock depicted in Figure 7.10.

a. Show that the four necessary conditions for deadlock indeed hold in this example.

b. State a simple rule for avoiding deadlocks in this system.

Answer:

a. The four necessary conditions for a deadlock are (1) mutual exclusion; (2) hold-and-wait; (3) no preemption; and (4) circular wait. The mutual exclusion condition holds since only one car can occupy a space in the roadway. Hold-and-wait occurs where a car holds onto its place in the roadway while it waits to advance in the roadway. A car cannot be removed (i.e. preempted) from its position in the roadway. Lastly, there is indeed a circular wait as each car is waiting for a subsequent car to advance. The circular wait condition is also easily observed from the graphic.

b. A simple rule that would avoid this traffic deadlock is that a car may not advance into an intersection if it is clear it will not be able immediately to clear the intersection.

7.12 Assume a multithreaded application uses only reader—writer locks for synchronization. Applying the four necessary conditions for deadlock, is deadlock still possible if multiple reader—writer locks are used?

Answer:

YES. (1) Mutual exclusion is maintained, as they cannot be shared if there is a writer. (2) Hold-and-wait is possible, as a thread can hold one reader —writer lock while waiting to acquire another. (3) You cannot take a lock away, so no preemption is upheld. (4) A circular wait among all threads is possible.

7.13 The program example shown in Figure 7.4 doesn't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.

Answer:

If `thread_one` is scheduled before `thread_two` and `thread_one` is able to acquire both mutex locks before `thread_two` is scheduled, deadlock will not occur. Deadlock can only occur if either `thread_one` or `thread_two` is able to acquire only one lock before the other thread acquires the second lock.

7.14 In Section 7.4.4, we describe a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if two threads simultaneously invoke the `transaction()` function. Fix the `transaction()` function to prevent deadlocks.

Answer:

Add a new lock to this function. This third lock must be acquired before the two locks associated with the accounts are acquired. The `transaction()` function now appears as follows:

```
void transaction(Account from, Account to, double amount)
{
  Semaphore lock1, lock2, lock3;
  wait(lock3);
  lock1 = getLock(from);
  lock2 = getLock(to);

  wait(lock1);
    wait(lock2);

      withdraw(from, amount);
      deposit(to, amount);

    signal(lock3);
    signal(lock2);
  signal(lock1);
}
```

7.15 Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:

a. Runtime overheads

b. System throughput

Answer:
A deadlock-avoidance scheme tends to increase the runtime overheads due to the cost of keep track of the current resource allocation. However, a deadlock-avoidance scheme allows for more concurrent use of resources than schemes that statically prevent the formation of deadlock. In that sense, a deadlock-avoidance scheme could increase system throughput.

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

a. Increase **Available** (new resources added)

b. Decrease **Available** (resource permanently removed from system)

c. Increase **Max** for one process (the process needs or wants more resources than allowed).

d. Decrease **Max** for one process (the process decides it does not need that many resources)

e. Increase the number of processes

f. Decrease the number of processes

Answer:

a. Increase **Available** (new resources added)—This could safely be changed without any problems.

b. Decrease **Available** (resource permanently removed from system) —This could have an effect on the system and introduce the possibility of deadlock as the safety of the system assumed there were a certain number of available resources.

c. Increase **Max** for one process (the process needs more resources than allowed, it may want more)—This could have an effect on the system and introduce the possibility of deadlock.

d. Decrease **Max** for one process (the process decides it does not need that many resources)—This could safely be changed without any problems.

e. Increase the number of processes—This could be allowed assuming that resources were allocated to the new process(es) such that the system does not enter an unsafe state.

f. Decrease the number of processes—This could safely be changed without any problems.

7.17 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

Answer:

Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources and, therefore it will return its resources when done.

7.18 Consider a system consisting of m resources of the same type being shared by n processes. A process can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

a. The maximum need of each process is between 1 and m resources

b. The sum of all maximum needs is less than $m + n$

Answer:

Using the terminology of Section Section 7.6.2, we have:

a. $\sum_{i=1}^{n} Max_i < m + n$

b. $Max_i \geq 1$ for all i
Proof: $Need_i = Max_i - Allocation_i$
If there exists a deadlock state then:

c. $\sum_{i=1}^{n} Allocation_i = m$

Use a. to get: $\sum Need_i + \sum Allocation_i = \sum Max_i < m + n$
Use c. to get: $\sum Need_i + m < m + n$
Rewrite to get: $\sum_{i=1}^{n} Need_i < n$
This implies that there exists a process P_i such that $Need_i = 0$. Since $Max_i \geq 1$ it follows that P_i has at least one resource that it can release. Hence the system cannot be in a deadlock state.

7.19 Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

Answer:

The following rule prevents deadlock: when a philosopher makes a request for the first chopstick, do not grant the request if there is no other philosopher with two chopsticks and if there is only one chopstick remaining.

7.20 Consider again the setting in the preceding question. Assume now that each philosopher requires three chopsticks to eat. Resource requests are still issued one at a time. Describe some simple rules for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

Answer:

When a philosopher makes a request for a chopstick, allocate the request if: 1) the philosopher has two chopsticks and there is at least one chopstick remaining, 2) the philosopher has one chopstick and there are at least two chopsticks remaining, 3) there is at least one chopstick remaining, and there is at least one philosopher with three chopsticks, 4) the philosopher has no chopsticks, there are two chopsticks remaining, and there is at least one other philosopher with two chopsticks assigned.

7.21 We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1. Show through an example that we cannot implement the multiple-resource-type banker's scheme by applying the single-resource-type scheme to each resource type individually.

Answer:

Consider a system with resources *A*, *B*, and *C* and processes P_0, P_1, P_2, P_3, and P_4 with the following values of *Allocation*:

Allocation			
	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

and the following value of *Need*:

Need			
	A	B	C
P_0	7	4	3
P_1	0	2	0
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

If the value of *Available* is (2 3 0), we can see that a request from process P_0 for (0 2 0) cannot be satisfied as this lowers *Available* to (2 1 0) and no process could safely finish.

However, if we treat the three resources as three single-resource types of the banker's algorithm, we get the following:
For resource *A* (of which we have 2 available),

	Allocated	Need
P_0	0	7
P_1	3	0
P_2	3	6
P_3	2	0
P_4	0	4

Processes could safely finish in the order P_1, P_3, P_4, P_2, P_0.

For resource B (of which we now have 1 available as 2 were assumed assigned to process P_0),

	Allocated	Need
P_0	3	2
P_1	0	2
P_2	0	0
P_3	1	1
P_4	0	3

Processes could safely finish in the order P_2, P_3, P_1, P_0, P_4.

And finally, for For resource C (of which we have 0 available),

	Allocated	Need
P_0	0	3
P_1	2	0
P_2	2	0
P_3	1	1
P_4	2	1

Processes could safely finish in the order P_1, P_2, P_0, P_3, P_4.

As we can see, if we use the banker's algorithm for multiple resource types, the request for resources (0 2 0) from process P_0 is denied as it leaves the system in an unsafe state. However, if we consider the banker's algorithm for the three separate resources where we use a single resource type, the request is granted. Therefore, if we have multiple resource types, we must use the banker's algorithm for multiple resource types.

7.22 Consider the following snapshot of a system:

	Allocation	**Max**
	A B C D	*A B C D*
P_0	3 0 1 4	5 1 1 7
P_1	2 2 1 0	3 2 1 1
P_2	3 1 2 1	3 3 2 1
P_3	0 5 1 0	4 6 1 2
P_4	4 2 1 2	6 3 2 5

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the processes may complete. Otherwise, illustrate why the state is unsafe.

a. **Available** = (0, 3, 0, 1)

b. **Available** = (1, 0, 0, 2)

Answer:

a. Not safe. Processes P_2, P_1, and P_3 are able to finish, but no remaining processes can finish.
Safe. Processes P_1, P_2, and P_3 are able to finish. Following this, processes P_0 and P_4 are also able to finish.

7.23 Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C D	*A B C D*	*A B C D*
P_0	2 0 0 1	4 2 1 2	3 3 2 1
P_1	3 1 2 1	5 2 5 2	
P_2	2 1 0 3	2 3 1 6	
P_3	1 3 1 2	1 4 2 4	
P_4	1 4 3 2	3 6 6 5	

Answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the processes may complete.

b. If a request from process P_1 arrives for (1, 1, 0, 0), can the request be granted immediately?

c. If a request from process P_4 arrives for (0, 0, 2, 0), can the request be granted immediately?

7.24 What is the optimistic assumption made in the deadlock-detection algorithm? How could this assumption be violated?
Answer:
The optimistic assumption is that there will not be any form of circular wait in terms of resources allocated and processes making requests for them. This assumption could be violated if a circular wait does indeed occur in practice.

7.25 A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if both a northbound and a southbound farmer get on the bridge at the same time (Vermont farmers are stubborn and are unable to back up). Using semaphores, design an algorithm that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, and vice versa).
Answer:

```
semaphore ok_to_cross = 1;

void enter_bridge() {
    ok_to_cross.wait();
}

void exit_bridge() {
    ok_to_cross.signal();
}
```

7.26 Modify your solution to Exercise 7.25 so that it is starvation-free.
Answer:

```
monitor bridge {
    int num_waiting_north = 0;
    int num_waiting_south = 0;
    int on_bridge = 0;
    condition ok_to_cross;
    int prev = 0;

    void enter_bridge_north() {
      num_waiting_north++;
      while (on_bridge ||
              (prev == 0 && num_waiting_south > 0))
      ok_to_cross.wait();
      num_waiting_north--;
      prev = 0;
    }

    void exit_bridge_north() {
      on_bridge = 0;
      ok_to_cross.broadcast();
    }

    void enter_bridge_south() {
      num_waiting_south++;
      while (on_bridge ||
              (prev == 1 && num_waiting_north > 0))
      ok_to_cross.wait();
      num_waiting_south--;
      prev = 1;
    }

    void exit_bridge_south() {
      on_bridge = 0;
      ok_to_cross.broadcast();
    }
}
```

Main Memory

Exercises

8.9 Explain the difference between internal and external fragmentation.
Answer:

a. Internal fragmentation is the area in a region or a page that is not used by the job occupying that region or page. This space is unavailable for use by the system until that job is finished and the page or region is released.

b. External fragmentation is unused space between allocated regions of memory. Typically external fragmentation results in memory regions that are too small to satisfy a memory request, but if we were to combine all the regions of external fragmentation, we would have enough memory to satisfy a memory request.

8.10 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linkage editor is used to combine multiple object modules into a single program binary. How does the linkage editor change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linkage editor to facilitate the memory-binding tasks of the linkage editor?
Answer:
The linkage editor has to replace unresolved symbolic addresses with the actual addresses associated with the variables in the final program binary. In order to perform this, the modules should keep track of instructions that refer to unresolved symbols. During linking, each module is assigned a sequence of addresses in the overall program binary and when this has been performed, unresolved references to symbols exported by this binary could be patched in other modules since every other module would contain the list of instructions that need to be patched.

8.11 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory.

Answer:

a. **First-fit**:
b. 115 KB is put in 300 KB partition, leaving (185 KB, 600 KB, 350 KB, 200 KB, 750 KB, 125 KB)
c. 500 KB is put in 600 KB partition, leaving (185 KB, 100 KB, 350 KB, 200 KB, 750 KB, 125 KB)
d. 358 KB is put in 750 KB partition, leaving (185 KB, 100 KB, 350 KB, 200 KB, 392 KB, 125 KB)
e. 200 KB is put in 350 KB partition, leaving (185 KB, 100 KB, 150 KB, 200 KB, 392 KB, 125 KB)
f. 375 KB is put in 392 KB partition, leaving (185 KB, 100 KB, 150 KB, 200 KB, 17 KB, 125 KB)
g. **Best-fit**:
h. 115 KB is put in 125 KB partition, leaving (300 KB, 600 KB, 350 KB, 200 KB, 750 KB, 10 KB)
i. 500 KB is put in 600 KB partition, leaving (300 KB, 100 KB, 350 KB, 200 KB, 750 KB, 10 KB)
j. 358 KB is put in 750 KB partition, leaving (300 KB, 100 KB, 350 KB, 200 KB, 392 KB, 10 KB)
k. 200 KB is put in 200 KB partition, leaving (300 KB, 100 KB, 350 KB, 0 KB, 392 KB, 10 KB)
l. 375 KB is put in 392 KB partition, leaving (300 KB, 100 KB, 350 KB, 0 KB, 17 KB, 10 KB)
m. **Worst-fit**:
n. 115 KB is put in 750 KB partition, leaving (300 KB, 600 KB, 350 KB, 200 KB, 635 KB, 125 KB)
o. 500 KB is put in 635 KB partition, leaving (300 KB, 600 KB, 350 KB, 200 KB, 135 KB, 125 KB)
p. 358 KB is put in 600 KB partition, leaving (300 KB, 242 KB, 350 KB, 200 KB, 135 KB, 125 KB)
q. 200 KB is put in 350 KB partition, leaving (300 KB, 242 KB, 150 KB, 200 KB, 135 KB, 125 KB)
r. 375 KB must wait

In this example, only worst-fit does not allow a request to be satisfied. An argument could be made that best-fit is most efficient as it leaves the largest holes after allocation. However, best-fit runs at time $O(n)$ and first-fit runs in constant time $O(1)$.

8.12 Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Contiguous memory allocation

b. Pure segmentation

c. Pure paging

Answer:

a. contiguous-memory allocation: might require relocation of the entire program since there is not enough space for the program to grow its allocated memory space.

b. pure segmentation: might also require relocation of the segment that needs to be extended since there is not enough space for the segment to grow its allocated memory space.

c. pure paging: incremental allocation of new pages is possible in this scheme without requiring relocation of the program's address space.

8.13 Compare the memory organization schemes of contiguous memory allocation, pure segmentation, and pure paging with respect to the following issues:

a. External fragmentation

b. Internal fragmentation

c. Ability to share code across processes

Answer:
The contiguous memory allocation scheme suffers from external fragmentation as address spaces are allocated contiguously and holes develop as old processes die and new processes are initiated. It also does not allow processes to share code, since a process's virtual memory segment is not broken into noncontiguous finegrained segments. Pure segmentation also suffers from external fragmentation as a segment of a process is laid out contiguously in physical memory and fragmentation would occur as segments of dead processes are replaced by segments of new processes. Segmentation, however, enables processes to share code; for instance, two different processes could share a code segment but have distinct data segments. Pure paging does not suffer from external fragmentation, but instead suffers from internal fragmentation. Processes are allocated in page granularity and if a page is not completely utilized, it results in internal fragmentation and a corresponding wastage of space. Paging also enables processes to share code at the granularity of pages.

8.14 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to other memory? Why should it or should it not?
Answer:

An address on a paging system is a logical page number and an offset. The physical page is found by searching a table based on the logical page number to produce a physical page number. Because the operating system controls the contents of this table, it can limit a process to accessing only those physical pages allocated to the process. There is no way for a process to refer to a page it does not own because the page will not be in the page table. To allow such access, an operating system simply needs to allow entries for non-process memory to be added to the process's page table. This is useful when two or more processes need to exchange data—they just read and write to the same physical addresses (which may be at varying logical addresses). This makes for very efficient interprocess communication.

8.15 Explain why mobile operating systems such as iOS and Android do not support swapping.
Answer: There are three reasons: First is that these mobile devices typically use flash memory with limited capacity and swapping is avoided because of this space constraint. Second, flash memory can support a limited number of write operations before it becomes less reliable. Lastly, there is typically poor throughput between main memory and flash memory.

8.16 Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?
Answer: Primarily because Android does not wish for its boot disk to be used as swap space for the reasons outlined in the previous question – the boot disk has limited storage capacity. However, Android does support swapping, it is just that users must provide their own separate SD card for swap space.

8.17 Compare paging with segmentation with respect to how much memory the address translation structures require to convert virtual addresses to physical addresses.
Answer:
Paging requires more memory overhead to maintain the translation structures. Segmentation requires just two registers per segment: one to maintain the base of the segment and the other to maintain the extent of the segment. Paging on the other hand requires one entry per page, and this entry provides the physical address in which the page is located.

8.18 Explain why address space identifiers (ASIDs) are used.
Answer: ASIDs provide address space protection in the TLB as well as supporting TLB entries for several different processes at the same time.

8.19 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment that is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

a. Contiguous memory allocation

b. Pure segmentation

c. Pure paging

Answer:
1) Contiguous-memory allocation requires the operating system to allocate the entire extent of the virtual address space to the program when it starts executing. This could be much larger than the actual memory requirements of the process. 2) Pure segmentation gives the operating system flexibility to assign a small extent to each segment at program startup time and extend the segment if required. 3) Pure paging does not require the operating system to allocate the maximum extent of the virtual address space to a process at startup time, but it still requires the operating system to allocate a large page table spanning all of the program's virtual address space. When a program needs to extend the stack or the heap, it needs to allocate a new page but the corresponding page table entry is preallocated.

8.20 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

Answer:

a. page = 3; offset = 13

b. page = 41; offset = 111

c. page = 210; offset = 161

d. page = 634; offset = 784

e. page = 1953; offset = 129

8.21 The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

Answer: Conventional, single-level page table will have $2^{10} = 1024$ entries. Inverted page table will have $2^5 = 32$ entries.

8.22 What is the maximum amount of physical memory in the BTV operating system?
Answer: $2^{16} = 65536$ (or 64-KB.)

8.23 Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

Answer:

a. $12 + 8 = 20$ bits.

b. $12 + 6 = 18$ bits.

8.24 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a. A conventional single-level page table

b. An inverted page table

Answer:

a. 2^{20} entries.

b. 512 K K/4K = 128K entries.

8.25 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

b. If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

Answer:

a. 400 nanoseconds: 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

b. Effective access time = 0.75 × (200 nanoseconds) + 0.25 × (400 nanoseconds) = 250 nanoseconds.

8.26 Why are segmentation and paging sometimes combined into one scheme?
Answer:
Segmentation and paging are often combined in order to improve upon each other. Segmented paging is helpful when the page table becomes very large. A large contiguous section of the page table that is unused can be collapsed into a single-segment table entry with a page-table address of zero. Paged segmentation handles the case of having very long segments that require a lot of time for allocation. By paging the

segments, we reduce wasted memory due to external fragmentation as well as simplify the allocation.

8.27 Explain why sharing a reentrant module is easier when segmentation is used than when pure paging is used.
Answer:
Since segmentation is based on a logical division of memory rather than a physical one, segments of any size can be shared with only one entry in the segment tables of each user. With paging there must be a common entry in the page tables for each page that is shared.

8.28 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430

b. 1,10

c. 2,500

d. 3,400

e. 4,112

Answer:

a. 219 + 430 = 649

b. 2300 + 10 = 2310

c. illegal reference, trap to operating system

d. 1327 + 400 = 1727

e. illegal reference, trap to operating system

8.29 What is the purpose of paging the page tables?
Answer:
In certain situations the page tables could become large enough that by paging the page tables, one could simplify the memory allocation problem (by ensuring that everything is allocated as fixed-size pages as opposed to variable-sized chunks) and also enable the swapping of portions of page table that are not currently used.

8.30 Consider the hierarchical paging scheme used by the VAX architecture. How many memory operations are performed when a user program executes a memory-load operation?
Answer:

When a memory load operation is performed, there are three memory operations that might be performed. One is to translate the position where the page table entry for the page could be found (since page tables themselves are paged). The second access is to access the page table entry itself, while the third access is the actual memory load operation.

8.31 Compare the segmented paging scheme with the hashed page table scheme for handling large address spaces. Under what circumstances is one scheme preferable to the other?

Answer:

When a program occupies only a small portion of its large virtual address space, a hashed page table might be preferred due to its smaller size. The disadvantage with hashed page tables however is the problem that arises due to conflicts in mapping multiple pages onto the same hashed page table entry. If many pages map to the same entry, then traversing the list corresponding to that hash table entry could incur a significant overhead; such overheads are minimal in the segmented paging scheme where each page table entry maintains information regarding only one page.

8.32 Consider the Intel address-translation scheme shown in Figure 8.22.

a. Describe all the steps taken by the Intel Pentium in translating a logical address into a physical address.

b. What are the advantages to the operating system of hardware that provides such complicated memory translation?

c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?

Answer:

a. The selector is an index into the segment descriptor table. The segment descriptor result plus the original offset is used to produce a linear address with a dir, page, and offset. The dir is an index into a page directory. The entry from the page directory selects the page table, and the page field is an index into the page table. The entry from the page table, plus the offset, is the physical address.

b. Such a page-translation mechanism offers the flexibility to allow most operating systems to implement their memory scheme in hardware, instead of having to implement some parts in hardware and some in software. Because it can be done in hardware, it is more efficient (and the kernel is simpler).

c. Address translation can take longer due to the multiple table lookups it can invoke. Caches help, but there will still be cache misses.

Virtual Memory

Virtual memory can be a very interesting subject since it has so many different aspects: page faults, managing the backing store, page replacement, frame allocation, thrashing, page size. The objectives of this chapter are to explain these concepts and show how paging works.

A simulation is probably the easiest way to allow the students to program several of the page-replacement algorithms and see how they really work. If an interactive graphics display can be used to display the simulation as it works, the students may be better able to understand how paging works. We also present an exercise that asks the student to develop a Java program that implements the FIFO and LRU page-replacement algorithms.

Exercises

9.14 Assume a program has just referenced an address in virtual memory. Describe a scenario how each of the following can occur: (If a scenario cannot occur, explain why.)

- TLB miss with no page fault
- TLB miss and page fault
- TLB hit and no page fault
- TLB hit and page fault

Answer:

- TLB miss with no page fault page has been brought into memory, but has been removed from the TLB
- TLB miss and page fault page fault has occurred
- TLB hit and no page fault page is in memory and in the TLB. Most likely a recent reference
- TLB hit and page fault cannot occur. The TLB is a cache of the page table. If an entry is not in the page table, it will not be in the TLB.

9.15 A simplified view of thread states is **Ready**, **Running**, and **Blocked**, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O). This is illustrated in Figure 9.31. Assuming a thread is in the Running state, answer the following questions, and explain your answer:

a. Will the thread change state if it incurs a page fault? If so, to what new state?

b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what new state?

c. Will the thread change state if an address reference is resolved in the page table? If so, to what new state?

Answer:

- On a page fault the thread state is set to blocked as an I/O operation is required to bring the new page into memory.
- On a TLB-miss, the thread continues running if the address is resolved in the page table.
- The thread will continue running if the address is resolved in the page table.

9.16 Consider a system that uses pure demand paging:

a. When a process first start execution, how would you characterize the page fault rate?

b. Once the working set for a process is loaded into memory, how would you characterize the page fault rate?

c. Assume a process changes its locality and the size of the new working set is too large to be stored into available free memory. Identify some options system designers could choose from to handle this situation?

Answer:

a. Initially quite high as needed pages are not yet loaded into memory.

b. It should be quite low as all necessary pages are loaded into memory.

c. (1) Ignore it; (2) get more physical memory; (3) reclaim pages more aggressively due to the high page fault rate.

9.17 What is the copy-on-write feature, and under what circumstances is it beneficial? What hardware support is required to implement this feature?

Answer:

When two processes are accessing the same set of program values (for instance, the code segment of the source binary), then it is useful to map the corresponding pages into the virtual address spaces of the two programs in a write-protected manner. When a write does indeed take place, then a copy must be made to allow the two programs to individually access the different copies without interfering with each other. The hardware support required to implement is simply the following: on each memory access, the page table needs to be consulted to check whether the page is write protected. If it is indeed write protected, a trap would occur and the operating system could resolve the issue.

9.18 A certain computer provides its users with a virtual-memory space of 2^{32} bytes. The computer has 2^{18} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.
Answer:
The virtual address in binary form is

0001 0001 0001 0010 0011 0100 0101 0110

Since the page size is 2^{12}, the page table size is 2^{20}. Therefore the low-order 12 bits "0100 0101 0110" are used as the displacement into the page, while the remaining 20 bits "0001 0001 0001 0010 0011" are used as the displacement in the page table.

9.19 Assume we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty page is available or the replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?
Answer:

$$\begin{aligned} 0.2\ \mu\text{sec} &= (1-P)\times 0.1\ \mu\text{sec} + (0.3P)\times 8\ \text{millisec} + (0.7P)\times 20\ \text{millisec} \\ 0.1 &= -0.1P + 2400\ P + 14000\ P \\ 0.1 &\simeq 16{,}400\ P \\ P &\simeq 0.000006 \end{aligned}$$

9.20 When a page fault occurs, the process requesting the page must block while waiting for the page to be brought from disk into physical memory. Assume that there exists a process with five user-level threads and that the mapping of user threads to kernel threads is many to one. If one user thread incurs a page fault while accessing its stack, will the other user threads belonging to the same process also be affected by the page fault—that is, will they also have to wait for the faulting page to be brought into memory? Explain.
Answer:
Yes, because there is only one kernel thread for all user threads, that kernel thread blocks while waiting for the page fault to be resolved.

Since there are no other kernel threads for available user threads, all other user threads in the process are thus affected by the page fault.

9.21 Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

Answer:

- 18
- 17
- 13

9.22 The following page table is for a system with 16-bit virtual and physical addresses and with 4,096-byte pages. The reference bit is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

Page	Page Frame	Reference Bit
0	9	0
1	1	0
2	14	0
3	10	0
4	–	0
5	13	0
6	8	0
7	15	0
8	–	0
9	0	0
10	5	0
11	4	0
12	–	0
13	–	0
14	3	0
15	2	0

a. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.

- `0xE12C`
- `0x3A9D`
- `0xA9D9`
- `0x7001`
- `0xACA1`

b. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.

c. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?

Answer:

- ○ `0xE12C` → `0x312C`
  ○ `0x3A9D` → `0xAA9D`
  ○ `0xA9D9` → `0x59D9`
  ○ `0x7001` → `0xF001`
  ○ `0xACA1` → `0x5CA1`
- The only choices are pages 4, 8, 12, and 13. Thus, example addresses include anything that begins with the hexadecimal sequence `0x4...`, `0x8...`, `0xC...`, and `0xD...`.
- Any page table entries that have a reference bit of zero. This includes the following frames {9, 1, 14, 13, 8, 0, 4}

9.23 Assume that you are monitoring the rate at which the pointer in the clock algorithm (which indicates the candidate page for replacement) moves. What can you say about the system if you notice the following behavior:

a. pointer is moving fast

b. pointer is moving slow

Answer:
If the pointer is moving fast, then the program is accessing a large number of pages simultaneously. It is most likely that during the period between the point at which the bit corresponding to a page is cleared and it is checked again, the page is accessed again and therefore cannot be replaced. This results in more scanning of the pages before a victim page is found. If the pointer is moving slow, then the virtual memory system is finding candidate pages for replacement extremely efficiently, indicating that many of the resident pages are not being accessed.

9.24 Discuss situations in which the LFU page-replacement algorithm generates fewer page faults than the LRU page-replacement algorithm. Also discuss under what circumstances the opposite holds.

Answer:
Consider the following sequence of memory accesses in a system that can hold four pages in memory: 1 1 2 3 4 5 1. When page 5 is accessed, the least frequently used page-replacement algorithm would replace a page other than 1, and therefore would not incur a page fault when page 1 is accessed again. On the other hand, for the sequence "1 2 3 4 5 2," the least recently used algorithm performs better.

9.25 Discuss situations in which the MFU page-replacement algorithm generates fewer page faults than the LRU page-replacement algorithm. Also discuss under what circumstances the opposite holds.
Answer:
Consider the sequence in a system that holds four pages in memory: 1 2 3 4 4 4 5 1. The most frequently used page replacement algorithm evicts page 4 while fetching page 5, while the LRU algorithm evicts page 1. This is unlikely to happen much in practice. For the sequence "1 2 3 4 4 4 5 1," the LRU algorithm makes the right decision.

9.26 The VAX/VMS system uses a FIFO replacement algorithm for resident pages and a free-frame pool of recently used pages. Assume that the free-frame pool is managed using the least recently used replacement policy. Answer the following questions:

a. If a page fault occurs and if the page does not exist in the free-frame pool, how is free space generated for the newly requested page?

b. If a page fault occurs and if the page exists in the free-frame pool, how is the resident page set and the free-frame pool managed to make space for the requested page?

c. What does the system degenerate to if the number of resident pages is set to one?

d. What does the system degenerate to if the number of pages in the free-frame pool is zero?

Answer:

a. When a page fault occurs and if the page does not exist in the free-frame pool, then one of the pages in the free-frame pool is evicted to disk, creating space for one of the resident pages to be moved to the free-frame pool. The accessed page is then moved to the resident set.

b. When a page fault occurs and if the page exists in the free-frame pool, then it is moved into the set of resident pages, while one of the resident pages is moved to the free-frame pool.

c. When the number of resident pages is set to one, then the system degenerates into the page replacement algorithm used in the free-frame pool, which is typically managed in a LRU fashion.

d. When the number of pages in the free-frame pool is zero, then the system degenerates into a FIFO page-replacement algorithm.

9.27 Consider a demand-paging system with the following time-measured utilizations:

CPU utilization	20%
Paging disk	97.7%
Other I/O devices	5%

For each of the following, say whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU.

b. Install a bigger paging disk.

c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

f. Install a faster hard disk or multiple controllers with multiple hard disks.

g. Add prepaging to the page fetch algorithms.

h. Increase the page size.

Answer:
The system obviously is spending most of its time paging, indicating over-allocation of memory. If the level of multiprogramming is reduced resident processes would page fault less frequently and the CPU utilization would improve. Another way to improve performance would be to get more physical memory or a faster paging drum.

a. Install a faster CPU—No.

b. Install a bigger paging disk—No.

c. Increase the degree of multiprogramming—No.

d. Decrease the degree of multiprogramming—Yes.

e. Install more main memory—Likely to improve CPU utilization as more pages can remain resident and not require paging to or from the disks.

f. Install a faster hard disk or multiple controllers with multiple hard disks—Also an improvement, for as the disk bottleneck is removed by faster response and more throughput to the disks, the CPU will get more data more quickly.

g. Add prepaging to the page fetch algorithms—Again, the CPU will get more data faster, so it will be more in use. This is only the case if the paging action is amenable to prefetching (i.e., some of the access is sequential).

h. Increase the page size—Increasing the page size will result in fewer page faults if data is being accessed sequentially. If data access is more or less random, more paging action could ensue because fewer pages can be kept in memory and more data is transferred per page fault. So this change is as likely to decrease utilization as it is to increase it.

9.28 Suppose that a machine provides instructions that can access memory locations using the one-level indirect addressing scheme. What is the sequence of page faults incurred when all of the pages of a program are currently non resident and the first instruction of the program is an indirect memory load operation? What happens when the operating system is using a per-process frame allocation technique and only two pages are allocated to this process?

Answer:

The following page faults take place: page fault to access the instruction, a page fault to access the memory location that contains a pointer to the target memory location, and a page fault when the target memory location is accessed. The operating system will generate three page faults with the third page replacing the page containing the instruction. If the instruction needs to be fetched again to repeat the trapped instruction, then the sequence of page faults will continue indefinitely. If the instruction is cached in a register, then it will be able to execute completely after the third page fault.

9.29 Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

Answer:

Such an algorithm could be implemented with the use of a reference bit. After every examination, the bit is set to zero; set back to one if the page is referenced. The algorithm would then select an arbitrary page for replacement from the set of unused pages since the last examination.

The advantage of this algorithm is its simplicity—nothing other than a reference bit need be maintained. The disadvantage of this algorithm is that it ignores locality by using only a short time frame for determining whether to evict a page or not. For example, a page may be part of the working set of a process, but may be evicted because it was not referenced since the last examination (that is, not all pages in the working set may be referenced between examinations).

9.30 A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then, to replace a page, we can search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address these problems:

   i. What the initial value of the counters is

ii. When counters are increased
iii. When counters are decreased
iv. How the page to be replaced is selected

b. How many page faults occur for your algorithm for the following reference string, for four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

Answer:

a. Define a page-replacement algorithm addressing the problems of:
   i. Initial value of the counters—0.
   ii. Counters are increased—whenever a new page is associated with that frame.
   iii. Counters are decreased—whenever one of the pages associated with that frame is no longer required.
   iv. How the page to be replaced is selected—find a frame with the smallest counter. Use FIFO for breaking ties.

b. 14 page faults

c. 11 page faults

9.31 Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

Answer:

$$\begin{aligned}\text{effective access time} &= (0.8) \times (1\ \mu\text{sec}) \\ &\quad + (0.1) \times (2\ \mu\text{sec}) + (0.1) \times (5002\ \mu\text{sec}) \\ &= 501.2\ \mu\text{sec} \\ &= 0.5\ \text{millisec}\end{aligned}$$

9.32 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

Answer:

Thrashing is caused by underallocation of the minimum number of pages required by a process, forcing it to continuously page fault. The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

9.33 Is it possible for a process to have two working sets, one representing data and another representing code? Explain.

Answer:

Yes, in fact many processors provide two TLBs for this very reason. As an example, the code being accessed by a process may retain the same working set for a long period of time. However, the data the code accesses may change, thus reflecting a change in the working set for data accesses.

9.34 Consider the parameter Δ used to define the working-set window in the working-set model. What is the effect of setting Δ to a small value on the page fault frequency and the number of active (non-suspended) processes currently executing in the system? What is the effect when Δ is set to a very high value?

Answer:

When Δ is set to a small value, then the set of resident pages for a process might be underestimated, allowing a process to be scheduled even though all of its required pages are not resident. This could result in a large number of page faults. When Δ is set to a large value, then a process's resident set is overestimated and this might prevent many processes from being scheduled even though their required pages are resident. However, once a process is scheduled, it is unlikely to generate page faults since its resident set has been overestimated.

9.35 Assume there is an initial 1024 KB segment where memory is allocated using the Buddy system. Using Figure Figure 9.26 as a guide, draw the tree illustrating how the following memory requests are allocated:

- request 240 bytes
- request 120 bytes
- request 60 bytes
- request 130 bytes

Next, modify the tree for the following releases of memory. Perform coalescing whenever possible:

- release 240 bytes
- release 60 bytes
- release 120 bytes

Answer:

The following allocation is made by the Buddy system: The 240-byte request is assigned a 256-byte segment. The 120-byte request is assigned a 128-byte segement, the 60-byte request is assigned a 64-byte segment and the 130-byte request is assigned a 256-byte segment. After the allocation, the following segment sizes are available: 64-bytes, 256-bytes, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, and 512K.

After the releases of memory, the only segment in use would be a 256-byte segment containing 130 bytes of data. The following segments will be free: 256 bytes, 512 bytes, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, and 512K.

9.36 A system provides support for user-level and kernel-level threads. The mapping in this system is one to one (there is a corresponding kernel thread for each user thread). Does a multithreaded process consist of (a) a working set for the entire process or (b) a working set for each thread? Explain.

Answer:

A working set for each thread. This is because each kernel thread has its own execution sequence, thus generating its unique sequence of addresses.

9.37 The slab allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this doesn't scale well with multiple CPUs. What could be done to address this scalability issue?

Answer:

This has long been a problem with the slab allocator—poor scalability with multiple CPUs. The issue comes from having to lock the global cache when it is being access. This has the effect of serializing cache accesses on multiprocessor systems. Solaris has addressed this by introducing a per-CPU cache, rather than a single global cache.

9.38 Consider a system that allocates pages of different sizes to its processes. What are the advantages of such a paging scheme? What modifications to the virtual memory system provide this functionality?

Answer:

The program could have a large code segment or use large-sized arrays as data. These portions of the program could be allocated to larger pages, thereby decreasing the memory overheads associated with a page table. The virtual memory system would then have to maintain multiple free lists of pages for the different sizes and also needs to have more complex code for address translation to take into account different page sizes.

Mass Storage Structure

CHAPTER 10

In this chapter we describe the internal data structures and algorithms used by the operating system to implement the file system. We also discuss the lowest level of the file system, the secondary storage structure. We first describe disk-head-scheduling algorithms. Next we discuss disk formatting and management of boot blocks, damaged blocks, and swap space. We end with coverage of disk reliability and stable storage.

The basic implementation of disk scheduling should be fairly clear: requests, queues, servicing; so the main new consideration is the actual algorithms: FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK. Simulation may be the best way to involve the student with the algorithms.

The paper by Worthington et al. [1994] gives a good presentation of the disk-scheduling algorithms and their evaluation. Be suspicious of the results of the disk-scheduling papers from the 1970s, such as Teory and Pinkerton [1972], because they generally assume that the seek time function is linear, rather than a square root. The paper by Lynch [1972b] shows the importance of keeping the overall system context in mind when choosing scheduling algorithms. Unfortunately, it is fairly difficult to find.

Chapter 10 introduced the concept of primary, secondary, and tertiary storage. In this chapter, we discuss tertiary storage in more detail. First, we describe the types of storage devices used for tertiary storage. Next, we discuss the issues that arise when an operating system uses tertiary storage. Finally, we consider some performance aspects of tertiary storage systems.

Exercises

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

a. Explain why this assertion is true.

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

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

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

Answer:

a. New requests for the track over which the head currently resides can theoretically arrive as quickly as these requests are being serviced.

b. All requests older than some predetermined age could be "forced" to the top of the queue, and an associated bit for each could be set to indicate that no new request could be moved ahead of these requests. For SSTF, the rest of the queue would have to be reorganized with respect to the last of these "old" requests.

c. To prevent unusually long response times.

d. Paging and swapping should take priority over user requests. It may be desirable for other kernel-initiated I/O, such as the writing of file system metadata, to take precedence over user I/O. If the kernel supports real-time process priorities, the I/O requests of those processes should be favored.

10.11 Explain why SSDs often use a FCFS disk scheduling algorithm.
Answer:
Because SSDs do not have moving parts and therefore performance is insensitive to issues such as seek time and rotational latency. Therefore, a simple FCFS policy will suffice.

10.12 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 2150, and the previous request was at cylinder 1805. The queue of pending requests, in FIFO order, is:

2069, 1212, 2296, 2800, 544, 1618, 356, 1523, 4965, 3681

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS

b. SSTF

c. SCAN

d. LOOK

e. C-SCAN

f. C-LOOK

Answer:

a. FILL

b. FILL

c. FILL

d. FILL

e. FILL

f. FILL

10.13 Elementary physics states that when an object is subjected to a constant acceleration a, the relationship between distance d and time t is given by $d = \frac{1}{2}at^2$. Suppose that, during a seek, the disk in Exercise 10.11 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond and a full-stroke seek over all 5000 cylinders in 18 milliseconds.

a. The distance of a seek is the number of cylinders that the head moves. Explain why the seek time is proportional to the square root of the seek distance.

b. Write an equation for the seek time as a function of the seek distance. This equation should be of the form $t = x + y\sqrt{L}$, where t is the time in milliseconds and L is the seek distance in cylinders.

c. Calculate the total seek time for each of the schedules in Exercise 10.11. Determine which schedule is the fastest (has the smallest total seek time).

d. The *percentage speedup* is the time saved divided by the original time. What is the percentage speedup of the fastest schedule over FCFS?

Answer:

a. Solving $d = \frac{1}{2}at^2$ for t gives $t = \sqrt{(2d/a)}$.

b. Solve the simultaneous equations $t = x+y\sqrt{L}$ that result from ($t = 1$, $L = 1$) and ($t = 18$, $L = 4999$) to obtain $t = 0.7561 + 0.2439\sqrt{L}$.

c. The total seek times are: FCFS 65.20; SSTF 31.52; SCAN 62.02; LOOK 40.29; C-SCAN 62.10 (and C-LOOK 40.42). Thus, SSTF is fastest here.

d. $(65.20 - 31.52)/65.20 = 0.52$. The percentage speedup of SSTF over FCFS is 52%, with respect to the seek time. If we include the overhead of rotational latency and data transfer, the percentage speedup will be less.

10.14 Suppose that the disk in Exercise 10.12 rotates at 7200 RPM.

a. What is the average rotational latency of this disk drive?

b. What seek distance can be covered in the time that you found for part a?

Answer:

a. 7200 rpm gives 120 rotations per second. Thus, a full rotation takes 8.33 ms, and the average rotational latency (a half rotation) takes 4.167 ms.

b. Solving $t = 0.7561 + 0.2439\sqrt{L}$ for $t = 4.167$ gives $L = 195.58$, so we can seek over 195 tracks (about 4% of the disk) during an average rotational latency.

10.15 Describe some advantages and disadvantages of using SSDs as a caching tier and as a disk drive replacement compared to a system with just magnetic disks.
Answer:
SSDs have the advantage of being faster than magnetic disks as there are no moving parts and therefore do not have seek time or rotational latency.

10.16 Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?
Answer:
There is no simple analytical argument to answer the first part of this question. It would make a good small simulation experiment for the students. The answer can be found in Figure 2 of Worthington et al. [1994]. (Worthington et al. studied the LOOK algorithm, but similar results obtain for SCAN.) Figure 2 in Worthington et al. shows that C-LOOK has an average response time just a few percent higher than LOOK but that C-LOOK has a significantly lower variance in response time for medium and heavy workloads. The intuitive reason for the difference in variance is that LOOK (and SCAN) tend to favor requests near the middle cylinders, whereas the C-versions do not have this imbalance. The intuitive reason for the slower response time of C-LOOK is the "circular" seek from one end of the disk to the farthest request at the other end. This seek satisfies no requests. It causes only a small performance degradation because the square-root dependency of seek time on distance implies that a long seek isn't terribly expensive by comparison with moderate-length seeks.
For the second part of the question, we observe that these algorithms do not schedule to improve rotational latency; therefore, as seek times decrease relative to rotational latency, the performance differences between the algorithms will decrease.

10.17 Requests are not usually uniformly distributed. For example, we can expect a cylinder containing the file-system metadata to be accessed more frequently than a cylinder containing only files. Suppose you know that 50 percent of the requests are for a small, fixed number of cylinders.

a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.

b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this "hot spot" on the disk.

Answer:

a. SSTF would take greatest advantage of the situation. FCFS could cause unnecessary head movement if references to the "high-demand" cylinders were interspersed with references to cylinders far away.

b. Here are some ideas. Place the hot data near the middle of the disk. Modify SSTF to prevent starvation. Add the policy that if the disk becomes idle for more than, say, 50 ms, the operating system generates an *anticipatory seek* to the hot region, since the next request is more likely to be there.

10.18 Consider a RAID Level 5 organization comprising five disks, with the parity for sets of four blocks on four disks stored on the fifth disk. How many blocks are accessed in order to perform the following?

a. A write of one block of data

b. A write of seven continuous blocks of data

Answer:

a. A write of one block of data requires the following: read of the parity block, read of the old data stored in the target block, computation of the new parity based on the differences between the new and old contents of the target block, and write of the parity block and the target block.

b. Assume that the seven contiguous blocks begin at a four-block boundary. A write of seven contiguous blocks of data could be performed by writing the seven contiguous blocks, writing the parity block of the first four blocks, reading the eight block, computing the parity for the next set of four blocks and writing the corresponding parity block onto disk.

10.19 Compare the throughput achieved by a RAID Level 5 organization with that achieved by a RAID Level 1 organization for the following:

a. Read operations on single blocks

b. Read operations on multiple contiguous blocks

Answer:

a. The amount of throughput depends on the number of disks in the RAID system. A RAID Level 5 comprising of a parity block for every set of four blocks spread over five disks can support four to five operations simultaneously. A RAID Level 1 comprising of two disks can support two simultaneous operations. Of course, there is greater flexibility in RAID Level 1 as to which copy of a block could be accessed and that could provide performance benefits by taking into account position of disk head.

b. RAID Level 5 organization achieves greater bandwidth for accesses to multiple contiguous blocks since the adjacent blocks could be simultaneously accessed. Such bandwidth improvements are not possible in RAID Level 1.

10.20 Compare the performance of write operations achieved by a RAID Level 5 organization with that achieved by a RAID Level 1 organization.

Answer:

RAID Level 1 organization can perform writes by simply issuing the writes to mirrored data concurrently. RAID Level 5, on the other hand, would require the old contents of the parity block to be read before it is updated based on the new contents of the target block. This results in more overhead for the write operations on a RAID Level 5 system.

10.21 Assume that you have a mixed configuration comprising disks organized as RAID Level 1 and as RAID Level 5 disks. Assume that the system has flexibility in deciding which disk organization to use for storing a particular file. Which files should be stored in the RAID Level 1 disks and which in the RAID Level 5 disks in order to optimize performance?

Answer:

Frequently updated data need to be stored on RAID Level 1 disks while data that is more frequently read as opposed to being written should be stored in RAID Level 5 disks.

10.22 The reliability of a hard-disk drive is typically described in terms of a quantity called *mean time between failures* (*MTBF*). Although this quantity is called a "time," the MTBF actually is measured in drive-hours per failure.

a. If a system contains 1000 drives, each of which has a 750,000-hour MTBF, which of the following best describes how often a drive failure will occur in that disk farm: once per thousand years, once per century, once per decade, once per year, once per month, once per week, once per day, once per hour, once per minute, or once per second?

b. Mortality statistics indicate that, on the average, a U.S. resident has about 1 chance in 1000 of dying between ages 20 and 21 years. Deduce the MTBF hours for 20 year olds. Convert this figure from hours to years. What does this MTBF tell you about the expected lifetime of a 20 year old?

c. The manufacturer guarantees a 1-million-hour MTBF for a certain model of disk drive. What can you conclude about the number of years for which one of these drives is under warranty?

Answer:

a. 750,000 drive-hours per failure divided by 1000 drives gives 750 hours per failure—about 31 days or once per month.

b. The human-hours per failure is 8760 (hours in a year) divided by 0.001 failure, giving a value of 8,760,000 "hours" for the MTBF.

8760,000 hours equals 1000 years. This tells us nothing about the expected lifetime of a person of age 20.

c. The MTBF tells nothing about the expected lifetime. Hard disk drives are generally designed to have a lifetime of five years. If such a drive truly has a million-hour MTBF, it is very unlikely that the drive will fail during its expected lifetime.

10.23 Discuss the relative advantages and disadvantages of sector sparing and sector slipping.

Answer:

Sector sparing can cause an extra track switch and rotational latency, causing an unlucky request to require an extra 8 ms of time. Sector slipping has less impact during future reading, but at sector remapping time it can require the reading and writing of an entire track's worth of data to slip the sectors past the bad spot.

10.24 Discuss the reasons why the operating system might require accurate information on how blocks are stored on a disk. How could the operating system improve file system performance with this knowledge?

Answer:

While allocating blocks for a file, the operating system could allocate blocks that are geometrically close by on the disk if it had more information regarding the physical location of the blocks on the disk. In particular, it could allocate a block of data and then allocate the second block of data in the same cylinder but on a different surface at a rotationally optimal place so that the access to the next block could be made with minimal cost.

File-System Interface

Files are the most obvious object that operating systems manipulate. Everything is typically stored in files: programs, data, output, etc. The student should learn what a file is to the operating system and what the problems are (providing naming conventions to allow files to be found by user programs, protection).

Two problems can crop up in this chapter. First, terminology may be different between your system and the book. This can be used to drive home the point that concepts are important and terms must be clearly defined when you get to a new system. Second, it may be difficult to motivate students to learn about directory structures that are not the ones on the system they are using. This can best be overcome if the students have two very different systems to consider, such as a single-user system for a microcomputer and a large, university time-shared system.

Projects might include a report about the details of the file system for the local system. It is also possible to write programs to implement a simple file system either in memory (allocate a large block of memory that is used to simulate a disk) or on top of an existing file system. In many cases, the design of a file system is an interesting project of its own.

Exercises

11.9 Consider a file system where a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

Answer:

Let F1 be the old file and F2 be the new file. A user wishing to access F1 through an existing link will actually access F2. Note that the access protection for file F1 is used rather than the one associated with F2.

This problem can be avoided by insuring that all links to a deleted file are deleted also. This can be accomplished in several ways:

a. maintain a list of all links to a file, removing each of them when the file is deleted

b. retain the links, removing them when an attempt is made to access a deleted file

c. maintain a file reference list (or counter), deleting the file only after all links or references to that file have been deleted

11.10 The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate table for each user or just maintain one table that contains references to files that are being accessed by all users at the current time? If the same file is being accessed by two different programs or users, should there be separate entries in the open file table?

Answer:

By keeping a central open-file table, the operating system can perform the following operation that would be infeasible otherwise. Consider a file that is currently being accessed by one or more processes. If the file is deleted, then it should not be removed from the disk until all processes accessing the file have closed it. This check can be performed only if there is centralized accounting of number of processes accessing the file. On the other hand, if two processes are accessing the file, then two separate states need to be maintained to keep track of the current location of which parts of the file are being accessed by the two processes. This requires the operating system to maintain separate entries for the two processes.

11.11 What are the advantages and disadvantages of a system providing mandatory locks instead of providing advisory locks whose usage is left to the users' discretion?

Answer:

In many cases, separate programs might be willing to tolerate concurrent access to a file without requiring the need to obtain locks and thereby guaranteeing mutual exclusion to the files. Mutual exclusion could be guaranteed by other program structures such as memory locks or other forms of synchronization. In such situations, the mandatory locks would limit the flexibility in how files could be accessed and might also increase the overheads associated with accessing files.

11.12 Provide examples of applications that typically access files according to the following methods:

- Sequential
- Random

Answer:

- Applications that access files sequentially include word processors, music players, video players, and web servers.
- Applications that access files randomly include databases, video and audio editors.

11.13 Some systems automatically open a file when it is referenced for the first time, and close the file when the job terminates. Discuss the advantages

and disadvantages of this scheme as compared to the more traditional one, where the user has to open and close the file explicitly.
Answer:
Automatic opening and closing of files relieves the user from the invocation of these functions, and thus makes it more convenient to the user; however, it requires more overhead than the case where explicit opening and closing is required.

11.14 If the operating system were to know that a certain application is going to access the file data in a sequential manner, how could it exploit this information to improve performance?
Answer:
When a block is accessed, the file system could prefetch the subsequent blocks in anticipation of future requests to these blocks. This prefetching optimization would reduce the waiting time experienced by the process for future requests.

11.15 Give an example of an application that could benefit from operating system support for random access to indexed files.
Answer:
An application that maintains a database of entries could benefit from such support. For instance, if a program is maintaining a student database, then accesses to the database cannot be modeled by any predetermined access pattern. The access to records are random and locating the records would be more efficient if the operating system were to provide some form of tree-based index.

11.16 Discuss the advantages and disadvantages of supporting links to files that cross mount points (that is, the file link refers to a file that is stored in a different volume).
Answer:
The advantage is that there is greater transparency in the sense that the user does not need to be aware of mount points and create links in all scenarios. The disadvantage however is that the file system containing the link might be mounted while the file system containing the target file might not be, and therefore one cannot provide transparent access to the file in such a scenario; the error condition would expose to the user that a link is a dead link and that the link does indeed cross file system boundaries.

11.17 Some systems provide file sharing by maintaining a single copy of a file; other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.
Answer:
With a single copy, several concurrent updates to a file may result in user obtaining incorrect information, and the file being left in an incorrect state. With multiple copies, there is storage waste and the various copies may not be consistent with respect to each other.

11.18 Discuss the advantages and disadvantages of associating with remote file systems (stored on file servers) a different set of failure semantics from that associated with local file systems.

Answer:

The advantage is that the application can deal with the failure condition in a more intelligent manner if it realizes that it incurred an error while accessing a file stored in a remote file system. For instance, a file open operation could simply fail instead of hanging when accessing a remote file on a failed server and the application could deal with the failure in the best possible manner; if the operation were simply to hang, then the entire application hangs, which is not desirable. The disadvantage however is the lack of uniformity in failure semantics and the resulting complexity in application code.

11.19 What are the implications of supporting UNIX consistency semantics for shared access for those files that are stored on remote file systems?

Answer:

UNIX consistency semantics requires updates to a file to be immediately available to other processes. Supporting such a semantics for shared files on remote file systems could result in the following inefficiencies: all updates by a client have to be immediately reported to the file server instead of being batched (or even ignored if the updates are to a temporary file), and updates have to be communicated by the file server to clients caching the data immediately, again resulting in more communication.

File-System Implementation

In this chapter we discuss various methods for storing information on secondary storage. The basic issues are device directory, free space management, and space allocation on a disk.

Exercises

12.9 Consider a file system that uses a modifed contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?

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

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

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

Answer:
If all extents are of the same size, and the size is predetermined, then it simplifies the block allocation scheme. A simple bit map or free list for extents would suffice. If the extents can be of any size and are allocated dynamically, then more complex allocation schemes are required. It might be difficult to find an extent of the appropriate size and there might be external fragmentation. One could use the Buddy system allocator discussed in the previous chapters to design an appropriate allocator. When the extents can be of a few fixed sizes, and these sizes are predetermined, one would have to maintain a separate bitmap or free list for each possible size. This scheme is of intermediate complexity and of intermediate flexibility in comparison to the earlier schemes.

12.10 Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

Answer:

- Contiguous Sequential - Works very well as the file is stored contiguously.
- Sequential access - Simply involves traversing the contiguous disk blocks.
- Contiguous Random - Works very well as you can easily determine the adjacent disk block containing the position you wish to seek to.
- Linked Sequential - Satisfactory as you are simply following the links from one block to the next.
- Linked Random - Poor as it may require following the links to several disk blocks until you arrive at the intended seek point of the file.
- Indexed Sequential - Works well as sequential access simply involves sequentially accessing each index.
- Indexed Random - Works well as it is easy to determine the index associated with the disk block containing the position you wish to seek to

12.11 What are the advantages of the variation of linked allocation that uses a FAT to chain together the blocks of a file?
Answer:
The advantage is that while accessing a block that is stored at the middle of a file, its location can be determined by chasing the pointers stored in the FAT as opposed to accessing all of the individual blocks of the file in a sequential manner to find the pointer to the target block. Typically, most of the FAT can be cached in memory and therefore the pointers can be determined with just memory accesses instead of having to access the disk blocks.

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

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

b. Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at */a/b/c*? Assume that none of the disk blocks is currently being cached.

c. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

Answer:

a. In order to reconstruct the free list, it would be necessary to perform “garbage collection.” This would entail searching the entire directory structure to determine which pages are already allocated to jobs. Those remaining unallocated pages could be relinked as the free-space list.

b. Reading the contents of the small local file */a/b/c* involves 4 separate disk operations: (1) Reading in the disk block containing the root directory /, (2) & (3) reading in the disk block containing the directories *b* and *c*, and reading in the disk block containing the file *c*.

c. The free-space list pointer could be stored on the disk, perhaps in several places.

12.13 Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4 KB of disk space as a single 4-KB block or as eight 512-byte blocks. How could we take advantage of this flexibility to improve performance? What modifications would have to be made to the free-space management scheme in order to support this feature?

Answer:

Such a scheme would decrease internal fragmentation. If a file is 5 KB, then it could be allocated a 4 KB block and two contiguous 512-byte blocks. In addition to maintaining a bitmap of free blocks, one would also have to maintain extra state regarding which of the subblocks are currently being used inside a block. The allocator would then have to examine this extra state to allocate subblocks and coalesce the subblocks to obtain the larger block when all of the subblocks become free.

12.14 Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.

Answer:

The primary difficulty that might arise is due to delayed updates of data and metadata. Updates could be delayed in the hope that the same data might be updated in the future or that the updated data might be temporary and might be deleted in the near future. However, if the system were to crash without having committed the delayed updates, then the consistency of the file system is destroyed.

12.15 Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

Answer:

Let Z be the starting file address (block number).

- **Contiguous**. Divide the logical address by 512 with X and Y the resulting quotient and remainder respectively.

a. Add X to Z to obtain the physical block number. Y is the displacement into that block.

b. 1

- **Linked**. Divide the logical physical address by 511 with X and Y the resulting quotient and remainder respectively.

  a. Chase down the linked list (getting $X + 1$ blocks). $Y + 1$ is the displacement into the last physical block.

  b. 4

- **Indexed**. Divide the logical address by 512 with X and Y the resulting quotient and remainder respectively.

  a. Get the index block into memory. Physical block address is contained in the index block at location X. Y is the displacement into the desired physical block.

  b. 2

12.16 Consider a file system that uses inodes to represent files. Disk blocks are 8-KB in size and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, plus single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

Answer:

(12 * 8 /KB/) + (2048 * 8 /KB) + (2048 * 2048 * 8 /KB/) + (2048 * 2048 * 2048 * 8 /KB) = 64 terabytes

12.17 Fragmentation on a storage device could be eliminated by recompaction of the information. Typical disk devices do not have relocation or base registers (such as are used when memory is to be compacted), so how can we relocate files? Give three reasons why recompacting and relocation of files often are avoided.

Answer:

Relocation of files on secondary storage involves considerable overhead —data blocks have to be read into main memory and written back out to their new locations. Furthermore, relocation registers apply only to *sequential* files, and many disk files are not sequential. For this same reason, many new files will not require contiguous disk space; even sequential files can be allocated noncontiguous blocks if links between logically sequential blocks are maintained by the disk system.

12.18 Assume that in a particular augmentation of a remote-file-access protocol, each client maintains a name cache that caches translations from file names to corresponding file handles. What issues should we take into account in implementing the name cache?

Answer:

One issue is maintaining consistency of the name cache. If the cache entry becomes inconsistent, then either it should be updated or its inconsistency should be detected when it is used next. If the inconsistency is detected later, then there should be a fallback mechanism for determining the new translation for the name. Also, another related issue is whether a name lookup is performed one element at a time for each subdirectory in the pathname or whether it is performed in a single shot at the server. If it is perfomed one element at a time, then the client might obtain more information regarding the translations for all of the intermediate directories. On the other hand, it increases the network traffic as a single name lookup causes a sequence of partial name lookups.

12.19 Explain why logging metadata updates ensures recovery of a file system after a file system crash.

Answer:

For a file system to be recoverable after a crash, it must be consistent or must be able to be made consistent. Therefore, we have to prove that logging metadata updates keeps the file system in a consistent or able-to-be-consistent state. For a file system to become inconsistent, the metadata must be written incompletely or in the wrong order to the file system data structures. With metadata logging, the writes are made to a sequential log. The complete transaction is written there before it is moved to the file system structures. If the system crashes during file system data updates, the updates can be completed based on the information in the log. Thus, logging ensures that file system changes are made completely (either before or after a crash). The order of the changes is guaranteed to be correct because of the sequential writes to the log. If a change was made incompletely to the log, it is discarded, with no changes made to the file system structures. Therefore, the structures are either consistent or can be trivially made consistent via metadata logging replay.

12.20 Consider the following backup scheme:

- **Day 1**. Copy to a backup medium all files from the disk.
- **Day 2**. Copy to another medium all files changed since day 1.
- **Day 3**. Copy to another medium all files changed since day 1.

This differs from the schedule given in Section 12.7.4 by having all subsequent backups copy all files modified since the first full backup. What are the benefits of this system over the one in Section 12.7.4? What are the drawbacks? Are restore operations made easier or more difficult? Explain your answer.

Answer:

Restores are easier because you can go to the last backup tape, rather than the full tape. No intermediate tapes need be read. More tape is used as more files change.

I/O Systems

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. Although related topics appear in other chapters, here we bring together the pieces to paint a complete picture. In this chapter we describe I/O structure, devices, device drivers, caching, and terminal I/O.

Exercises

13.8 When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts.

Answer:

A number of issues need to be considered in order to determine the priority scheme to be used to determine the order in which the interrupts need to be serviced. First, interrupts raised by devices should be given higher priority than traps generated by the user program; a device interrupt can therefore interrupt code used for handling system calls. Second, interrupts that control devices might be given higher priority than interrupts that simply perform tasks such as copying data served up a device to user/kernel buffers, since such tasks can always be delayed. Third, devices that have real-time constraints on when their data is handled should be given higher priority than other devices. Also, devices that do not have any form of buffering for its data would have to be assigned higher priority since the data could be available only for a short period of time.

13.9 What are the advantages and disadvantages of supporting memory-mapped I/O to device-control registers?

Answer:

The advantage of supporting memory-mapped I/O to device-control registers is that it eliminates the need for special I/O instructions from the instruction set and therefore also does not require the enforcement of protection rules that prevent user programs from executing these I/O instructions. The disadvantage is that the resulting flexibility needs to be handled with care; the memory translation units need to ensure that the memory addresses associated with the device control registers are not accessible by user programs in order to ensure protection.

13.10 Consider the following I/O scenarios on a single-user PC.

a. A mouse used with a graphical user interface

b. A tape drive on a multitasking operating system (assume no device preallocation is available)

c. A disk drive containing user files

d. A graphics card with direct bus connection, accessible through memory-mapped I/O

For each of these I/O scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O, or interrupt-driven I/O? Give reasons for your choices.

Answer:

a. A mouse used with a graphical user interface
   Buffering may be needed to record mouse movement during times when higher-priority operations are taking place. Spooling and caching are inappropriate. Interrupt-driven I/O is most appropriate.

b. A tape drive on a multitasking operating system (assume no device preallocation is available)
   Buffering may be needed to manage throughput difference between the tape drive and the source or destination of the I/O. Caching can be used to hold copies of data that resides on the tape, for faster access. Spooling could be used to stage data to the device when multiple users desire to read from or write to it. Interrupt-driven I/O is likely to allow the best performance.

c. A disk drive containing user files
   Buffering can be used to hold data while in transit from user space to the disk, and visa versa. Caching can be used to hold disk-resident data for improved performance. Spooling is not necessary because disks are shared-access devices. Interrupt-driven I/O is best for devices such as disks that transfer data at slow rates.

d. A graphics card with direct bus connection, accessible through memory-mapped I/O
   Buffering may be needed to control multiple access and for performance (double-buffering can be used to hold the next screen image while displaying the current one). Caching and spooling are not necessary due to the fast and shared-access natures of the device. Polling and interrupts are useful only for input and for I/O completion detection, neither of which is needed for a memory-mapped device.

13.11 In most multiprogrammed systems, user programs access memory through virtual addresses, while the operating system uses raw physical addresses to access memory. What are the implications of this design on the initiation of I/O operations by the user program and their execution by the operating system?

Answer:

The user program typically specifies a buffer for data to be transmitted to or from a device. This buffer exists in user space and is specified by a virtual address. The kernel needs to issue the I/O operation and needs to copy data between the user buffer and its own kernel buffer before or after the I/O operation. In order to access the user buffer, the kernel needs to translate the virtual address provided by the user program to the corresponding physical address within the context of the user program's virtual address space. This translation is typically performed in software and therefore incurs overhead. Also, if the user buffer is not currently present in physical memory, the corresponding page(s) need to obtained from the swap space. This operation might require careful handling and might delay the data copy operation.

13.12 What are the various kinds of performance overheads associated with servicing an interrupt?

Answer:

When an interrupt occurs the currently executing process is interrupted and its state is stored in the appropriate process control block. The interrupt service routine is then dispatched in order to deal with the interrupt. On completion of handling of the interrupt, the state of the process is restored and the process is resumed. Therefore, the performance overheads include the cost of saving and restoring process state and the cost of flushing the instruction pipeline and restoring the instructions into the pipeline when the process is restarted.

13.13 Describe three circumstances under which blocking I/O should be used. Describe three circumstances under which nonblocking I/O should be used. Why not just implement nonblocking I/O and have processes busy-wait until their device is ready?

Answer:

Generally, blocking I/O is appropriate when the process will be waiting only for one specific event. Examples include a disk, tape, or keyboard read by an application program. Non-blocking I/O is useful when I/O may come from more than one source and the order of the I/O arrival is not predetermined. Examples include network daemons listening to more than one network socket, window managers that accept mouse movement as well as keyboard input, and I/O-management programs, such as a copy command that copies data between I/O devices. In the last case, the program could optimize its performance by buffering the input and output and using non-blocking I/O to keep both devices fully occupied.

Non-blocking I/O is more complicated for programmers, because of the asynchronous rendezvous that is needed when an I/O occurs. Also, busy waiting is less efficient than interrupt-driven I/O so the overall system performance would decrease.

13.14 Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces. The first piece executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

Answer:

The purpose of this strategy is to ensure that the most critical aspect of the interrupt handling code is performed first and the less critical portions of the code are delayed for the future. For instance, when a device finishes an I/O operation, the device-control operations corresponding to declaring the device as no longer being busy are more important in order to issue future operations. However, the task of copying the data provided by the device to the appropriate user or kernel memory regions can be delayed for a future point when the CPU is idle. In such a scenario, a lower-priority interrupt handler is used to perform the copy operation.

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

Answer:

Direct virtual memory access allows a device to perform a transfer from two memory-mapped devices without the intervention of the CPU or the use of main memory as a staging ground; the device simply issues memory operations to the memory-mapped addresses of a target device and the ensuing virtual address translation guarantees that the data is transferred to the appropriate device. This functionality, however, comes at the cost of having to support virtual address translation on addresses accessed by a DMA controller and requires the addition of an address-translation unit to the DMA controller. The address translation results in both hardware and software costs and might also result in coherence problems between the data structures maintained by the CPU for address translation and corresponding structures used by the DMA controller. These coherence issues would also need to be dealt with and results in further increase in system complexity.

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

Answer:

Three pros of the UNIX method: Very efficient, low overhead and low amount of data movement. Fast implementation—no coordination needed with other kernel components. Simple, so less chance of data loss

Three cons: No data protection, and more possible side effects from changes so more difficult to debug. Difficult to implement new I/O methods: new data structures needed rather than just new objects. Complicated kernel I/O subsystem, full of data structures, access routines, and locking mechanisms

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

Answer:

Each channel would run the following algorithm:

```
// A list of interrupts sorted in earliest-time-first
    order
List interruptList

// the list that associates a request with an entry
    in interruptList
List requestList

// an interrupt-based timer
Timer timer

while (true) {
    // Get the next earliest time in the list
    timer.setTime = interruptList.next();

    // An interrupt will occur at time timer.setTime

    //now wait for the timer interrupt
    i.e. for the timer to expire

   notify( requestList.next() );
}
```

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

Answer:

Reliable transfer of data requires modules to check whether space is available on the target module and to block the sending module if space is not available. This check requires extra communication between the modules, but the overhead enables the system to provide a stronger abstraction than one which does not guarantee reliable transfer. The stronger abstraction typically reduces the complexity of the code in the modules. In the STREAMS abstraction, however, there is unreliability introduced by the driver end, which is allowed to drop messages if the corresponding device cannot handle the incoming data. Consequently, even if there is reliable transfer of data between the modules, messages could be dropped at the device if the corresponding buffer fills up. This requires retransmission of data and special code for handling such retransmissions, thereby somewhat limiting the advantages that are associated with reliable transfer between the modules.

Protection

The various processes in an operating system must be protected from one another's activities. For that purpose, various mechanisms exist that can be used to ensure that the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

In this chapter, we examine the problem of protection in great detail and develop a unifying model for implementing protection.

It is important that the student learn the concepts of the access matrix, access lists, and capabilities. Capabilities have been used in several modern systems and can be tied in with abstract data types. The paper by Lampson [1971] is the classic reference on protection.

Exercises

14.11 Consider the ring protection scheme in MULTICS. If we were to implement the system calls of a typical operating system and store them in a segment associated with ring 0, what should be the values stored in the ring field of the segment descriptor? What happens during a system call when a process executing in a higher-numbered ring invokes a procedure in ring 0?

Answer:

The ring should be associated with an access bracket (b1, b2), a limit value b3, and a set of designated entry points. The processes that are allowed to invoke any code stored in segment 0 in an unconstrained manner are those processes that are currently executing in ring i where $b1 \leq i \leq b2$. Any other process executing within ring $b2 < i \leq b3$ is allowed to invoke only those functions that are designated entry points. This implies that we should have $b1 = 0$ and set $b2$ to be the highest ring number that comprises of system code that is allowed to invoke the code in segment 0 in an unconstrained fashion. We should also store only the system call functions as designated entry points and we should set $b3$ to be the ring number associated with user code so that user code can invoke the system calls.

14.12 The access-control matrix could be used to determine whether a process can switch from, say, domain A to domain B and enjoy the access privileges of domain B. Is this approach equivalent to including the access privileges of domain B in those of domain A?
Answer:
Yes, this approach is equivalent to including the access privileges of domain B in those of domain A as long as the switch privileges associated with domain B are also copied over to domain A.

14.13 Consider a computer system in which "computer games" can be played by students only between 10 P.M. and 6 A.M., by faculty members between 5 P.M. and 8 A.M., and by the computer center staff at all times. Suggest a scheme for implementing this policy efficiently.
Answer:
Set up a dynamic protection structure that changes the set of resources available with respect to the time allotted to the three categories of users. As time changes, so does the domain of users eligible to play the computer games. When the time comes that a user's eligibility is over, a revocation process must occur. Revocation could be immediate, selective (since the computer staff may access it at any hour), total, and temporary (since rights to access will be given back later in the day).

14.14 What hardware features does a computer system need for efficient capability manipulation? Can these features be used for memory protection?
Answer:
A hardware feature is needed allowing a capability object to be identified as either a capability of accessible object. Typically, several bits are necessary to distinguish between different types of capability objects. For example, 4 bits could be used to uniquely identify 2^4 or 16 different types of capability objects.

These could not be used for routine memory protection as they offer little else for protection apart from a binary value indicating whether they are a capability object or not. Memory protection requires full support from virtual memory features discussed in Chapter 9.

14.15 Discuss the strengths and weaknesses of implementing an access matrix using access lists that are associated with objects.
Answer:
The strength of storing an access list with each object is the control that comes from storing the access privileges along with each object, thereby allowing the object to revoke or expand the access privileges in a localized manner. The weakness with associating access lists is the overhead of checking whether the requesting domain appears on the access list. This check would be expensive and needs to be performed every time the object is accessed.

14.16 Discuss the strengths and weaknesses of implementing an access matrix using capabilities that are associated with domains.
Answer:
Capabilities associated with domains provide substantial flexibility and faster access to objects. When a domain presents a capability, the system

just needs to check the authenticity of the capability and that could be performed efficiently. Capabilities could also be passed around from one domain to another domain with great ease, allowing a system with a great amount of flexibility. However, the flexibility comes at the cost of a lack of control: revoking capabilities and restricting the flow of capabilities is a difficult task.

14.17 Explain why a capability-based system such as Hydra provides greater flexibility than the ring-protection scheme in enforcing protection policies.

Answer:

The ring-based protection scheme requires the modules to be ordered in a strictly hierarchical fashion. It also enforces the restriction that system code in internal rings cannot invoke operations in the external rings. This restriction limits the flexibility in structuring the code and is unnecessarily restrictive. The capability system provided by Hydra not only allows unstructured interactions between different modules, but also enables the dynamic instantiation of new modules as the need arises.

14.18 Discuss the need for rights amplification in Hydra. How does this practice compare with the cross-ring calls in a ring protection scheme?

Answer:

Rights amplification is required to deal with cross-domain calls where code in the calling domain does not have the access privileges to perform certain operations on an object but the called procedure has an expanded set of access privileges on the same object. Typically, when an object is created by a module, if the module wants to export the object to other modules without granting the other modules privileges to modify the object, it could export the object with those kinds of access privileges disabled. When the object is passed back to the module that created it in order to perform some mutations on it, the rights associated with the object need to be expanded. A more coarse-grained approach to rights amplification is employed in Multics. When a cross-ring call occurs, a set of checks are made to ensure that the calling code has sufficient rights to invoke the target code. Assuming that the checks are satisfied, the target code is invoked and the ring number associated with the process is modified to be ring number associated with the target code, thereby expanding the access rights associated with the process.

14.19 What is the need-to-know principle? Why is it important for a protection system to adhere to this principle?

Answer:

A process may access at any time those resources that it has been authorized to access *and* are required currently to complete its task. It is important in that it limits the amount of damage a faulty process can cause in a system.

14.20 Discuss which of the following systems allow module designers to enforce the need-to-know principle.

a. The MULTICS ring protection scheme

b. Hydra's capabilities

c. JVM's stack-inspection scheme

Answer:
The ring protection scheme in MULTICS does not necessarily enforce the need-to-know principle. If an object must be accessible in a domain at ring level j but not accessible in a domain at ring level i, then we must have $j < i$. But this requirement means that every object accessible in level i must also be accessible in level j.

A similar problem arises in JVM's stack inspection scheme. When a sequence of calls is made within a doPrivileged code block, then all of the code fragments in the called procedure have the same access privileges as the original code block that performed the doPrivileged operation, thereby violating the need-to-know principle.

In Hydra, the rights-amplification mechanism ensures that only the privileged code has access privileges to protected objects, and if this code were to invoke code in other modules, the objects could be exported to the other modules after lowering the access privileges to the exported objects. This mechanism provides fine-grained control over access rights and helps to guarantee that the need-to-know principle is satisfied.

14.21 Describe how the Java protection model would be sacrificed if a Java program were allowed to directly alter the annotations of its stack frame.

Answer:
When a thread issues an access request in a `doPrivileged()` block, the stack frame of the calling thread is *annotated* according to the calling thread's protection domain. A thread with an annotated stack frame can make subsequent method calls that require certain privileges. Thus, the annotation serves to mark a calling thread as being privileged. By allowing a Java program to directly alter the annotations of a stack frame, a program could potentially perform an operation for which it does not have the necessary permissions, thus violating the security model of Java.

14.22 How are the access-matrix facility and the role-based access-control facility similar? How do they differ?

Answer:
The roles in a role-based access control facility are similar to the domain in the access-matrix facility. Just as a domain is granted access to certain resources, a role is also granted access to the appropriate resources. The two approaches differ in the amount of flexibility and the kind of access privileges that are granted to the entities. Certain access-control facilities allow modules to perform a *switch* operation that allows them to assume the privileges of a different module, and this operation can be performed in a transparent manner. Such switches are less transparent in role-based systems where the ability to switch roles is not a privilege that is granted through a mechanism that is part of the access-control system, but instead requires the explicit use of passwords.

14.23 How does the principle of least privilege aid in the creation of protection systems?

Answer:

The principle of least privilege allows users to be given just enough privileges to perform their tasks. A system implemented within the framework of this principle has the property that a failure or compromise of a component does the minimum damage to the system since the failed or compromised component has the least set of privileges required to support its normal mode of operation.

14.24 How can systems that implement the principle of least privilege still have protection failures that lead to security violations?

Answer:

The principle of least privileges only limits the damage but does not prevent the misuse of access privileges associated with a module if the module were to be compromised. For instance, if a system code is given the access privileges to deal with the task of managing tertiary storage, a security loophole in the code would not cause any damage to other parts of the system, but it could still cause protection failures in accessing the tertiary storage.

CHAPTER

Security

The information stored in the system (both data and code), as well as the physical resources of the computer system, need to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. In this chapter, we examine the ways in which information may be misused or intentionally made inconsistent. We then present mechanisms to guard against this occurrence.

Exercises

15.1 Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.

Answer:

One form of hardware support that guarantees that a buffer-overflow attack does not take place is to prevent the execution of code that is located in the stack segment of a process's address space. Recall that buffer-overflow attacks are performed by overflowing the buffer on a stack frame and overwriting the return address of the function, thereby jumping to another portion of the stack frame that contains malicious executable code, that had been placed there as a result of the buffer overflow. By preventing the execution of code from the stack segment, this problem is eliminated.

Approaches that use a better programming methodology are typically built around the use of bounds-checking to guard against buffer overflows. Buffer overflows do not not occur in languages like Java where every array access is guaranteed to be within bounds through a software check. Such approaches require no hardware support but result in run-time costs associated with performing bounds-checking.

15.2 A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

Answer:
Whenever a user logs in, the system prints the last time that user was logged on the system.

15.3 What is the purpose of using a "salt" along with the user-provided password? Where should the "salt" be stored, and how should it be used?
Answer:
When a user creates a password, the system generates a random number (which is the salt) and appends it to the user-provided password, encrypts the resulting string and stores the encrypted result and the salt in the password file. When a password check is to be made, the password presented by the user is first concatenated with the salt and then encrypted before checking for equality with the stored password. Since the salt is different for different users, a password cracker cannot check a single candidate password, encrypt it, and check it against all of the encrypted passwords simultaneously.

15.4 The list of all passwords is kept within the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (Hint: Use different internal and external representations.)
Answer:
Encrypt the passwords internally so that they can only be accessed in coded form. The only person with access or knowledge of decoding should be the system operator.

15.5 An experimental addition to UNIX allows a user to connect a **watchdog** program to a file. The watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss two pros and two cons of using watchdogs for security.
Answer:
The watchdog program becomes the primary security mechanism for file access. Because of this we find its primary benefits and detractions. A benefit of this approach is that you have a centralized mechanism for controlling access to a file—the watchdog program. By ensuring the watchdog program has sufficient security techniques, you are assured of having secure access to the file. However, this is also the primary negative of this approach as well—the watchdog program becomes the bottleneck. If the watchdog program is not properly implemented (that is, it has a security hole), there are no other backup mechanisms for file protection.

15.6 The UNIX program COPS scans a given system for possible security holes and alerts the user to possible problems. What are two potential hazards of using such a system for security? How can these problems be limited or eliminated?
Answer:
The COPS program itself could be modified by an intruder to disable some of its features or even to take advantage of its features to create new security flaws. Even if COPS is not cracked, it is possible for an

intruder to gain a copy of COPS, study it, and locate security breaches which COPS does not detect. Then that intruder could prey on systems in which the management depends on COPS for security (thinking it is providing security), when all COPS is providing is management complacency. COPS could be stored on a read-only medium or file system to avoid its modification. It could be provided only to bona fide systems managers to prevent it from falling into the wrong hands. Neither of these is a foolproof solution, however.

15.7 Discuss a means by which managers of systems connected to the Internet could have designed their systems to limit or eliminate the damage done by a worm. What are the drawbacks of making the change that you suggest?
Answer:
"Firewalls" can be erected between systems and the Internet. These systems filter the packets moving from one side of them to the other, assuring that only valid packets owned by authorized users are allowed to access the protect systems. Such firewalls usually make use of the systems less convenient (and network connections less efficient).

15.8 Argue for or against the judicial sentence handed down against Robert Morris, Jr., for his creation and execution of the Internet worm discussed in Section 15.3.1.
Answer:
An argument against the sentence is that it was simply excessive. Furthermore, many have now commented that this worm actually made people more aware of potential vulnerabilities in the public Internet. An argument for the sentence is that this worm cost Internet users significant time and money and—considering its apparent intent—the sentence was appropriate.

We encourage professors to use a case such as this—and the many similar contemporary cases—as a topic for a class debate.

15.9 Make a list of six security concerns for a bank's computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.
Answer:

- In a protected location, well guarded: physical, human.
- Network tamperproof: physical, human, operating system.
- Modem access eliminated or limited: physical, human.
- Unauthorized data transfers prevented or logged: human, operating system.
- Backup media protected and guarded: physical, human.
- Programmers, data entry personnel, trustworthy: human.

15.10 What are two advantages of encrypting data stored in the computer system?

Answer:
Encrypted data are guarded by the operating system's protection facilities, as well as a password that is needed to decrypt them. Two keys are better than one when it comes to security.

15.11 What commonly used computer programs are prone to man-in-the-middle attacks? Discuss solutions for preventing this form of attack.
Answer:
Any protocol that requires a sender and a receiver to agree on a session key before they start communicating is prone to the man-in-the-middle attack. For instance, if one were to implement on a secure shell protocol by having the two communicating machines to identify a common session key, and if the protocol messages for exchanging the session key is not protected by the appropriate authentication mechanism, then it is possible for an attacker to manufacture a separate session key and get access to the data being communicated between the two parties. In particular, if the server is supposed to manufacture the session key, the attacker could obtain the session key from the server, communicate its locally manufactured session key to the client, and thereby convince the client to use the fake session key. When the attacker receives the data from the client, it can decrypt the data, reencrypt it with the original key from the server, and transmit the encrypted data to the server without alerting either the client or the server about the attacker's presence. Such attacks could be avoided by using digital signatures to authenticate messages from the server. If the server could communicate the session key and its identity in a message that is guarded by a digital signature granted by a certifying authority, then the attacker would not be able to forge a session key, and therefore the man-in-the-middle attack could be avoided.

15.12 Compare symmetric and asymmetric encryption schemes, and discuss under what circumstances a distributed system would use one or the other.
Answer:
A symmetric encryption scheme allows the same key to be used for encrypting and decrypting messages. An asymmetric scheme requires the use of two different keys for performing the encryption and the corresponding decryption. Asymmetric key cryptographic schemes are based on mathematical foundations that provide guarantees on the intractability of reverse-engineering the encryption scheme, but they are typically much more expensive than symmetric schemes, which do not provide any such theoretical guarantees. Asymmetric schemes are also superior to symmetric schemes since they could be used for other purposes such as authentication, confidentiality, and key distribution.

15.13 Why doesn't $D(k_d, N)(E(k_e, N)(m))$ provide authentication of the sender? To what uses can such an encryption be put?
Answer:
$D(k_d, N)(E(k_e, N)(m))$ means that the message is encrypted using the public key and then decrypted using the private key. This scheme is not sufficient to guarantee authentication since any entity can obtain the

public keys and therefore could have fabricated the message. However, the only entity that can decrypt the message is the entity that owns the private key, which guarantees that the message is a secret message from the sender to the entity owning the private key; no other entity can decrypt the contents of the message.

15.14 Discuss how the asymmetric encryption algorithm can be used to achieve the following goals.

a. Authentication: the receiver knows that only the sender could have generated the message.

b. Secrecy: only the receiver can decrypt the message.

c. Authentication and secrecy: only the receiver can decrypt the message, and the receiver knows that only the sender could have generated the message.

Answer:
Let k_e^s be the public key of the sender, k_e^r be the public key of the receiver, k_d^s be the private key of the sender, and k_e^s be the private key of the receiver. Authentication is performed by having the sender send a message that is encoded using k_d^s. Secrecy is ensured by having the sender encode the message using k_e^r. Both authentication and secrecy are guaranteed by performing double encryption using both k_d^s and k_e^r.

15.15 Consider a system that generates 10 million audit records per day. Also assume that there are on average 10 attacks per day on this system and that each such attack is reflected in 20 records. If the intrusion-detection system has a true-alarm rate of 0.6 and a false-alarm rate of 0.0005, what percentage of alarms generated by the system correspond to real intrusions?

Answer:
The probability of occurrence of intrusive records is $10 * 20/10^6 = 0.0002$. Using Bayes' theorem, the probability that an alarm corresponds to a real intrusion is simply $0.0002*0.6/(0.0002*0.6+0.9998*0.0005) = 0.193$.

Virtual Machines

The term *virtualization* has many meanings, and aspects of virtualization permeate all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware and that behaves toward them as native hardware would but that also protects, manages, and limits them.

This chapter delves into the uses, features, and implementation of virtual machines. Virtual machines can be implemented in several ways, and this chapter describes these options. One option is to add virtual machine support to the kernel. Because that implementation method is the most pertinent to this book, we explore it most fully. Additionally, hardware features provided by the CPU and even by I/O devices can support virtual machine implementation, so we discuss how those features are used by the appropriate kernel modules.

Exercises

16.1 Describe the three types of traditional virtualization.
Answer:

a. Type 0—implemented by firmware, low overhead but generally fewer features. Other VMMs can run as guests.

b. Type 1—special purpose software or general purpose operating systems that provide a means to run guests.Takes advantage of available hardware assistance, most feature rich.

c. Type 2—application providing guest execution unbeknownst to the operating system. More overhead and fewer features than type 1.

16.2 Describe the four virtualization-like execution environments and why they are not “true” virtualization.
Answer:
Paravirtualization - guest OS modified to perform better and integrate better with the VMM. Does not exactly duplicate native hardware. Programming-environment virtualization guest are programs written

in the programming language specific to the environment. Does not duplicate any real hardware, but rather an idealized theoretical system designed for the language. Emulation - runs via translating each instruction from a different CPU architecture to the current system's instructions. Runs too slowly to meet the definition of virtualization. Application containment - does not virtualization underlying hardware, but rather creates and environment for processes with many of the features of virtualization

16.3 Describe four benefits of virtualization.
Answer:

a. Consolidating multiple physical systems into guests on fewer physical systems - cost, power, cooling savings.

b. Easier management - X guest machines easier to monitor, administer than X physical systems.

c. Where available, live migration of guest between systems decreases downtime, eases administration, allows better resource management.

d. Development and QA - engineers have many operating systems and operating system versions available on a single system for development and testing.

16.4 Why can VMMs not implement trap-and-emulate-based virtualization on some CPUs? Lacking the ability to trap-and-emulate, what method can a VMM use to implement virtualization?
Answer:

a. If a CPU does not cause a trap if a privileged instruction is run in user mode, trap-and-emulate cannot work. For example an instruction could perform a subset of its function in user mode, and a full set in kernel mode. Such an instruction in guest mode does not cause a trap to the VMM to have its equivalent kernel mode effect.

b. The more complex and higher overhead binary translation method can be used in cases such as that.

16.5 What hardware assistance for virtualization can be provided by modern CPUs?
Answer:
VMMs in hardware provide CPU state storage of guests, nested page tables can be accelerated via table walking hardware, hardware assisted DMA allows hardware to transfer data directly into a guest memory space, and protection domains can limit which memory can be accessed by each guest for instances such as I/O data transfer.

16.6 Why is live migration possible in virtual environments but much less possible for a native operating system?
Answer:
Live migration moves a running process or guest from one server to another. A process has quite a lot of state data actively maintained by

the operating system, such as memory allocations, open files, network connections, and locks. Gathering and duplicating all of this state information while a process is executing is quite challenging and in fact is not done by any general purpose modern operating system. A guest on the other hand is very well contained, all of the sate of all of the processes within the guest is stored within the guest VM, with just a small amount of state about the guest (such as VCPU memory allocation, and network configuration) stored by the VMM. The VMM can send that information to the target host, send memory pages of the guest, then continue sending changed memory pages of the guest until the number changed per iteration is small. Finally the source VMM can pause the guest, send final changes, and once the target host acknowledges the transfer, delete the guest. The target host recreates the guest via these transfers and configures the guest network once the transfer is done such that all open connections to the guest now link to the target host rather than the source (assuming the networking hardware can allow the movement of a MAC address).

Distributed Systems

A *distributed system* is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication networks, such as high-speed buses or telephone lines. In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. We contrast the main differences in operating-system design between these types of systems and the centralized systems with which we were concerned previously.

The purpose of a DFS is to support the same kind of sharing when the files are physically dispersed among the various sites of a distributed system. We discuss the various ways a distributed file system can be designed and implemented. First, we discuss common concepts on which distributed file systems are based. Then, we illustrate our concepts by examining AFS —the Andrew distributed file system. By exploring this example system, we hope to provide a sense of the considerations involved in designing an operating system, and also to indicate current areas of operating-system research: network and distributed operating systems.

Exercises

17.9 What is the difference between computation migration and process migration? Which is easier to implement, and why?

Answer:

Process migration is an extreme form of computation migration. In computation migration, an RPC might be sent to a remote processor in order to execute a computation that could be more efficiently executed on the remote node. In process migration, the entire process is transported to the remote node, where the process continues its execution. Since process migration is an extension of computation migration, more issues need to be considered for implementing process migration. In particular, it is always challenging to migrate all of the necessary state to execute the process, and it is sometimes difficult to transport state regarding open files and open devices. Such a high degree of transparency and completeness is not required for computation migration, where it is clear to the programmer that only a certain section of the code is to be executed remotely. programmer.

17.10 Even though the ISO model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?

Answer:

A certain network layered protocol may achieve the same functionality of the ISO in fewer layers by using one layer to implement functionality provided in two (or possibly more) layers in the ISO model. Other models may decide there is no need for certain layers in the ISO model. For example, the presentation and session layers are absent in the TCP/IP protocol. Another reason may be that certain layers specified in the ISO model do not apply to a certain implementation. Let's use TCP/IP again as an example where no data link or physical layer is specified by the model. The thinking behind TCP/IP is that the functionality behind the data link and physical layers is not pertinent to TCP/IP—it merely assumes some network connection is provided, whether it be Ethernet, wireless, token ring, etc.

A potential problem with implementing fewer layers is that certain functionality may not be provided by features specified in the omitted layers.

17.11 Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance. What changes could help solve this problem?

Answer:

Faster systems may be able to send more packets in a shorter time. The network would then have more packets traveling on it, resulting in more collisions, and therefore less throughput relative to the number of packets being sent. More networks can be used, with fewer systems per network, to reduce the number of collisions.

17.12 What are the advantages of using dedicated hardware devices for routers and gateways? What are the disadvantages of using these devices compared with using general-purpose computers?

Answer:

The advantages are that dedicated hardware devices for routers and gateways are very fast as all their logic is provided in hardware (firmware.) Using a general-purpose computer for a router or gateway means that routing functionality is provided in software—which is not as fast as providing the functionality directly in hardware.

A disadvantage is that routers or gateways as dedicated devices may be more costly than using off-the-shelf components that comprise a modern personal computer.

17.13 In what ways is using a name server better than using static host tables? What problems or complications are associated with name servers? What methods could you use to decrease the amount of traffic name servers generate to satisfy translation requests?

Answer:
Name servers require their own protocol, so they add complication to the system. Also, if a name server is down, host information may become unavailable. Backup name servers are required to avoid this problem. Caches can be used to store frequently requested host information to cut down on network traffic.

17.14 Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?
Answer:
Hierarchical structures are easier to maintain since any changes in the identity of name servers require an update only at the next-level name server in the hierarchy. Changes are therefore localized. The downside of this approach, however, is that the name servers at the top level of the hierarchy are likely to suffer from high loads. This problem can be alleviated by replicating the services of the top-level name servers.

17.15 The lower layers of the ISO network model provide datagram service, with no delivery guarantees for messages. A transport-layer protocol such as TCP is used to provide reliability. Discuss the advantages and disadvantages of supporting reliable message delivery at the lowest possible layer.
Answer:
Many applications might not require reliable message delivery. For instance, a coded video stream could recover from packet losses by performing interpolations to derive lost data. In fact, in such applications, retransmitted data are of little use since they would arrive much later than the optimal time and not conform to realtime guarantees. For such applications, reliable message delivery at the lowest level is an unnecessary feature and might result in increased message traffic, most of which is useless, thereby resulting in performance degradation. In general, the lowest levels of the networking stack needs to support the minimal amount of functionality required by all applications and leave extra functionality to be implemented at the upper layers.

17.16 How does using a dynamic routing strategy affect application behavior? For what type of applications is it beneficial to use virtual routing instead of dynamic routing?
Answer:
Dynamic routing might route different packets through different paths. Consecutive packets might therefore incur different latencies and there could be substantial jitter in the received packets. Also, many protocols, such as TCP, that assume that reordered packets imply dropped packets, would have to be modified to take into account that reordering is a natural phenomenon in the system and does not imply packet losses. Realtime applications such as audio and video transmissions might benefit more from virtual routing since it minimizes jitter and packet reorderings.

17.17 Run the program shown in Figure 17.4 and determine the IP addresses of the following host names:

- www.wiley.com
- www.cs.yale.edu
- www.apple.com
- www.westminstercollege.edu
- www.ietf.org

Answer: As of August 2008, the corresponding IP addresses are

- www.wiley.com—`208.215.179.146`
- www.cs.yale.edu—`128.36.229.30`
- www.apple.com—`72.5.124.93`
- www.westminstercollege.edu—`146.86.1.17`
- www.ietf.org—`64.170.98.32`

17.18 The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?
Answer:
Despite the connectionless nature of UDP, it is not a serious alternative to TCP for the HTTP. The problem with UDP is that it is unreliable, documents delivered via the web must be delivered reliably. (This is easy to illustrate—a single packet missing from an image downloaded from the web makes the image unreadable.)

One possibility is to modify how TCP connections are used. Rather than setting up—and breaking down—a TCP connection for every web resource, allow *persistent* connections where a single TCP connection stays open and is used to deliver multiple web resources.

17.19 What are the advantages and the disadvantages of making the computer network transparent to the user?
Answer:
The advantage is that all files are accessed in the same manner. The disadvantage is that the operating system becomes more complex.

17.20 What are the benefits of a DFS when compared to a file system in a centralized system?
Answer:
A DFS allows the same type of sharing available on a centralized system, but the sharing may occur on physically and logically separate systems. Users around the world are able to share data as if they were in the same building, allowing a much more flexible computing environment than would otherwise be available.

17.21 Which of the example DFSs discussed in this chapter would handle a large, multiclient database application most efficiently? Explain your answer.

Answer:

The Andrew file system can handle a large, multiclient database as scalability is one of its hallmark features. Andrew is designed to handle up to 5,000 client workstations as well. A database also needs to run in a secure environment and Andrew uses the Kerberos security mechanism for encryption.

17.22 Discuss whether OpenAFS and NFS provide the following: (a) location transparency and (b) location independence.

Answer:

NFS provides location transparence since one cannot determine the server hosting the file from its name. (One could however view the mount-tables to determine the file server from which the corresponding file system is mounted, but the file server is not hardcoded in the name of the file.) NFS does not provide location independence since files cannot be moved automatically between different file systems. AFS provides location transparence and location independence.

17.23 Under what circumstances would a client prefer a location-transparent DFS? Under which circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.

Answer:

Location-transparent DFS is good enough in systems in which files are not replicated. Location-independent DFS is necessary when any replication is done.

17.24 What aspects of a distributed system would you select for a system running on a totally reliable network?

Answer:

Since the system is totally reliable, a stateful approach would make the most sense. Error recovery would seldom be needed, allowing the features of a stateful system to be used. If the network is very fast as well as reliable, caching can be done on the server side. On a slower network caching on both server and client will speed performance, as would file location-independence and migration. In addition, RPC-based service is not needed in the absence of failures, since a key part of its design is recovery during networking errors. Virtual-circuit systems are simpler and more appropriate for systems with no communications failures.

17.25 Consider OpenAFS, which is a stateful distributed file system. What actions need to be performed to recover from a server crash in order to preserve the consistency guaranteed by the system?

Answer:

A server needs to keep track of what clients are currently caching a file in order to issue a callback when the file is modified. When a server goes down, this state is lost. A server would then have to reconstruct this state typically by contacting all of the clients and having them report to the server what files are currently being cached by each client.

17.26 Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

Answer:

Caching locally can reduce network traffic substantially as the local cache can possibly handle a significant number of the remote accesses. This can reduce the amount of network traffic and lessen the load on the server. However, to maintain consistency, local updates to disk blocks must be updated on the server using either a write-through or delayed-write policy. A strategy must also be provided that allows the client to determine if its cached data is stale and needs to be updated.

Caching locally provides is obviously more complicated than having a client request all data from the server. But if access patterns indicate heavy writes to the data, the mechanisms for handling inconsistent data may increase network traffic and server load.

17.27 OpenAFS is designed to support a large number of clients. Discuss three techniques used to make OpenAFS a scalable system.

Answer:

Three techniques that make OpenAFS a scalable system are:

a. Caching: OpenAFS performs caching of files and name translations, thereby limiting the number of operations sent to the server.

b. Whole-file caching: when a file is opened, the entire contents of the file are transported to the client and no further interactions with the server are required. (This approach is refined in later versions where large chunks of a file rather than the entire file are transported in a single operation.)

c. Callbacks: It is the server's responsibility to revoke outdated copies of files. Clients can cache files and reuse the cached data multiple times without making requests to the server.

17.28 What are the benefits of mapping objects into virtual memory, as Apollo Domain does? What are the drawbacks?

Answer:

Mapping objects into virtual memory greatly eases the sharing of data between processes. Rather than opening a file, locking access to it, and reading and writing sections via the I/O system calls, memory-mapped objects are accessible as "normal" memory, with reads and writes to locations independent of disk pointers. Locking is much easier also, since one shared memory location can be used as a locking variable for semaphore access. Unfortunately, memory mapping adds complexity to the operating system, especially in a distributed system.

17.29 Describe some of the fundamental differences between OpenAFS and NFS (see Chapter 12).

Answer:

Some of the distinctive differences include:

a. OpenAFS has a rich set of features whereas NFS is organized around a much simpler design.

b. NFS allows a workstation to act as either a client, a server, or both. OpenAFS distinguishes between clients and server and identifies dedicated servers.

c. NFS is stateless, meaning a server does not maintain state during client updates of a file. OpenAFS is stateful between the period of when a client opens a file, updates it, and closes the file. (NFS does not even allow the opening and closing of files.)

d. Caching is a fundamental feature of OpenAFS, allowing client-side caching with cache consistency. In fact, it is an architectural principle behind OpenAFS to allow clients to cache entire files. Consistency is provided by servers when cached files are closed. The server then invalidates cached copies existing on other clients. Caching is allowed in NFS as well, but because of its stateless nature modified data must be committed to the server before results are received back by the client.

e. OpenAFS provides session semantics whereas NFS supports UNIX file consistency semantics.

The Linux System

Linux is a UNIX-like system that has gained popularity in recent years. In this chapter, we look at the history and development of Linux, and cover the user and programmer interfaces that Linux presents, interfaces that owe a great deal to the UNIX tradition. We also discuss the internal methods by which Linux implements these interfaces. However, since Linux has been designed to run as many standard UNIX applications as possible, it has much in common with existing UNIX implementations. We do not duplicate the basic description of UNIX given in the previous chapter.

Linux is a rapidly evolving operating system. This chapter describes specifically the Linux 2.6 kernel, released in late 2003.

Exercises

18.7 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?

Answer:

There are many advantages to writing an operating system in a high-level language such as C. First, by programming at a higher abstraction, the number of programming errors is reduced as the code becomes more compact. Second, many high-level languages provide advanced features such as bounds checking that further minimize programming errors and security loopholes. Also, high-level programming languages have powerful programming environments that include tools such as debuggers and performance profilers that could be handy for developing code. The disadvantage with using a high-level language is that the programmer is distanced from the underlying machine, which could cause a few problems. First, there could be a performance overhead introduced by the compiler and run-time system used for the high-level language. Second, certain operations and instructions that are available at the machine level might not be accessible from the language level, thereby limiting some of the functionality available to the programmer.

18.8 In what circumstances is the system-call sequence `fork()` `exec()` most appropriate? When is `vfork()` preferable?

Answer:

vfork() is a special case of clone and is used to create new processes without copying the page tables of the parent process. vfork() differs from fork in that the parent is suspended until the child makes a call to exec() or exit(). The child shares all memory with its parent, including the stack, until the child makes the call. This implies constraints on the program that it should be able to make progress without requiring the parent process to execute and is not suitable for certain programs where the parent and child processes interact before the child performs an exec. For such programs, the system-call sequence `fork()` `exec()` more appropriate.

18.9 What socket type should be used to implement an intercomputer file- transfer program? What type should be used for a program that periodically tests to see whether another computer is up on the network? Explain your answer.

Answer:

Sockets of type SOCK_STREAM use the TCP protocol for communicating data. The TCP protocol is appropriate for implementing an intercomputer file-transfer program since it provides a reliable, flow-controlled, and congestion-friendly communication channel. If data packets corresponding to a file transfer are lost, then they are retransmitted. Furthermore, the file transfer does not overrun buffer resources at the receiver and adapts to the available bandwidth along the channel. Sockets of type SOCK_DGRAM use the UDP protocol for communicating data. The UDP protocol is more appropriate for checking whether another computer is up on the network. Since a connection-oriented communication channel is not required and since there might not be any active entities on the other side to establish a communication channel with, the UDP protocol is more appropriate.

18.10 Linux runs on a variety of hardware platforms. What steps must the Linux developers take to ensure that the system is portable to different processors and memory-management architectures, and to minimize the amount of architecture-specific kernel code?

Answer:

The organization of architecture-dependent and architecture-independent code in the Linux kernel is designed to satisfy two design goals: to keep as much code as possible common between architectures and to provide a clean way of defining architecture-specific properties and code. The solution must of course be consistent with the overriding aims of code maintainability and performance.

There are different levels of architecture dependence in the kernel, and different techniques are appropriate in each case to comply with the design requirements. These levels include:

a. **CPU word size and endianness**. These are issues that affect the portability of all software written in C, but especially so for an operating system, where the size and alignment of data must be carefully arranged.

b. **CPU process architecture**. Linux relies on many forms of hardware support for its process and memory management. Different processors have their own mechanisms for changing between protection domains (e.g., entering kernel mode from user mode), rescheduling processes, managing virtual memory, and handling incoming interrupts.

The Linux kernel source code is organized so as to allow as much of the kernel as possible to be independent of the details of these architecture-specific features. To this end, the kernel keeps not one but two separate subdirectory hierarchies for each hardware architecture. One contains the code that is appropriate only for that architecture, including such functionality as the system call interface and low-level interrupt-management code.

The second architecture-specific directory tree contains C header files that are descriptive of the architecture. These header files contain type definitions and macros designed to hide the differences between architectures. They provide standard types for obtaining words of a given length, macro constants defining such things as the architecture word size or page size, and function macros to perform common tasks such as converting a word to a given byte order or doing standard manipulations to a page-table entry.

Given these two architecture-specific subdirectory trees, a large portion of the Linux kernel can be made portable between architectures. An attention to detail is required: when a 32-bit integer is required, the programmer must use the explicit __int32 type rather than assume than an int is a given size, for example. However, as long as the architecture-specific header files are used, then most process and page-table manipulation can be performed using common code between the architectures. Code that definitely cannot be shared is kept safely detached from the main common kernel code.

18.11 What are the advantages and disadvantages of making only some of the symbols defined inside a kernel accessible to a loadable kernel module?

Answer:

The advantage of making only some of the symbols defined inside a kernel accessible to a loadable kernel module is that there is a fixed set of entry points made available to the kernel module. This ensures that loadable modules cannot invoke arbitrary code within the kernel and thereby interfere with the kernel's execution. By restricting the set of entry points, the kernel is guaranteed that the interactions with the module take place at controlled points where certain invariants hold. The disadvantage of making only a small set of the symbols defined accessible to the kernel module is the loss in flexibility and might sometimes lead to a performance issue as some of the details of the kernel are hidden from the module.

18.12 What are the primary goals of the conflict resolution mechanism used by the Linux kernel for loading kernel modules?
Answer:
Conflict resolution prevents different modules from having conflicting access to hardware resources. In particular, when multiple drivers are trying to access the same hardware, it resolves the resulting conflict.

18.13 Discuss how the clone() operation supported by Linux is used to support both processes and threads.
Answer:
In Linux, threads are implemented within the kernel by a clone mechanism that creates a new process within the same virtual address space as the parent process. Unlike some kernel-based thread packages, the Linux kernel does not make any distinction between threads and processes: a thread is simply a process that did not create a new virtual address space when it was initialized.
The main advantage of implementing threads in the kernel rather than in a user-mode library are that:

- kernel-threaded systems can take advantage of multiple processors if they are available; and
- if one thread blocks in a kernel service routine (for example, a system call or page fault), other threads are still able to run.

18.14 Would one classify Linux threads as user-level threads or as kernel-level threads? Support your answer with the appropriate arguments.
Answer:
Linux threads are kernel-level threads. The threads are visible to the kernel and are independently schedule-able. User-level threads, on the other hand, are not visible to the kernel and are instead manipulated by user-level schedulers. In addition, the threads used in the Linux kernel are used to support both the thread abstraction and the process abstraction. A new process is created by simply associating a newly created kernel thread with a distinct address space, whereas a new thread is created by simply creating a new kernel thread with the same address space. This further indicates that the thread abstaction is intimately tied into the kernel.

18.15 What are the extra costs incurred by the creation and scheduling of a process, as compared to the cost of a cloned thread?
Answer:
In Linux, creation of a thread involves only the creation of some very simple data structures to describe the new thread. Space must be reserved for the new thread's execution context, its saved registers, its kernel stack page and dynamic information such as its security profile and signal state, but no new virtual address space is created.

Creating this new virtual address space is the most expensive part of the creation of a new process. The entire page table of the parent process must be copied, with each page being examined so that copy-on-write semantics can be achieved and so that reference counts to physical pages can be updated. The parent process's virtual memory is also affected by the process creation: any private read/write pages owned by the parent must be marked read-only so that copy-on-write can happen (copy-on-write relies on a page fault being generated when a write to the page occurs).

Scheduling of threads and processes also differs in this respect. The decision algorithm performed when deciding what process to run next is the same regardless of whether the process is a fully independent process or just a thread, but the action of context switching to a separate process is much more costly than switching to a thread. A process requires that the CPU's virtual memory control registers be updated to point to the new virtual address space's page tables.

In both cases—creation of a process or context switching between processes—the extra virtual memory operations have a significant cost. On many CPUs, changing page tables or swapping between page tables is not cheap: all or part of the virtual address translation look-aside buffers in the CPU must be purged when the page tables are changed. These costs are not incurred when creating or scheduling between threads.

18.16 How does Linux's Completely Fair Scheduler (CFS) provide improved fairness over a traditional UNIX process scheduler? When is the fairness guaranteed?

Answer:

The Completely Fair Scheduler (CFS) provides improved fairness over traditional process schedulers by assigning each process a proportion of the processor, instead of a fixed timeslice. CFSthus yields constant fairness but a variable switching rate. As the number of runnable processes on a system approaches infinity, the proportion of allotted processor approaches zero. To ensure that processes receive at least a minimum amount of the processor, CFSplaces a foor on the proportion of processor each process is allotted, called the **minimum granularity**. Thus CFSguarantees fairness only when the number of runnable processes is not so large that the proportion of allocated processor is floored by the minimum granularity. In the common case of only a handful of runnable processes, CFSis perfectly fair.

18.17 What are the Completely Fair Scheduler (CFS) two configurable variables? What are the pros and cons to setting each of them to very small and very large values?

Answer:

The Completely Fair Scheduler (CFS) provides two primary configuration knobs: **minimum granularity** and **target latency**. The minimum granularity (see also the previous question) sets a floor on the amount of processor that CFSallots each runnable process, minimizing switching costs at the expense of the fairness guarantee. Assigning a very small value will extend the fairness guarantee to a larger number of runnable processes, but will increase switching costs as those processes will each run for a very small amount of time. A very large value will yield the opposite: Processes will run for longer periods, but a relatively-smaller number of runnable processes will cause CFSto abdicate its fairness guarantee.

Target latency is the period in which all runnable processes will run, hence each process's scheduling latency is at most the target latency. For example, assume we have two runnable processes with a target latency of 20 milliseconds. Then each process will run for 10 milliseconds. Four processes, and each will run for 5 milliseconds. A very small target latency will amplify switching costs (and possibly run afoul of the minimum granularity). A very large target latency will minimize switching costs at the expensive of higher scheduling latency.

18.18 The Linux scheduler implements *soft* real-time scheduling. What features are missing that are necessary for some real-time programming tasks? How might they be added to the kernel?

Answer:

Linux's "soft" real-time scheduling provides ordering guarantees concerning the priorities of runnable processes: real-time processes will always be given a higher priority by the scheduler than normal time-sharing processes, and a real-time process will never be interrupted by another process with a lower real-time priority.

However, the Linux kernel does not support "hard" real-time functionality. That is, when a process is executing a kernel service routine, that routine will always execute to completion unless it yields control back to the scheduler either explicitly or implicitly (by waiting for some asynchronous event). There is no support for preemptive scheduling of kernel-mode processes. As a result, any kernel system call that runs for a significant amount of time without rescheduling will block execution of any real-time processes.

Many real-time applications require such hard real-time scheduling. In particular, they often require guaranteed worst-case response times to external events. To achieve these guarantees, and to give user-mode real-time processes a true higher priority than kernel-mode lower-priority processes, it is necessary to find a way to avoid having to wait for low-priority kernel calls to complete before scheduling a real-time process. For example, if a device driver generates an interrupt that wakes up a high-priority real-time process, then the kernel needs to be able to schedule that process as soon as possible, even if some other process is already executing in kernel mode.

Such preemptive rescheduling of kernel-mode routines comes at a cost. If the kernel cannot rely on non-preemption to ensure atomic updates of shared data structures, then reads of or updates to those structures must be protected by some other, finer-granularity locking mechanism. This fine-grained locking of kernel resources is the main requirement for provision of tight scheduling guarantees.

Many other kernel features could be added to support real-time programming. Deadline-based scheduling could be achieved by making modifications to the scheduler. Prioritization of IO operations could be implemented in the block-device IO request layer.

18.19 Under what circumstances would a user process request an operation that results in the allocation of a demand-zero memory region?
Answer:
Uninitialized data can be backed by demand-zero memory regions in a process's virtual address space. In addition, newly malloced space can also be backed by a demand-zero memory region.

18.20 What scenarios would cause a page of memory to be mapped into a user program's address space with the copy-on-write attribute enabled?
Answer:
When a process performs a fork operation, a new process is created based on the original binary but with a new address space that is a clone of the original address space. One possibility is to not to create a new address space but instead to share the address space between the old process and the newly created process. The pages of the address space are mapped with the copy-on-write attribute enabled. Then, when one of the processes performs an update on the shared address space, a new copy is made and the processes no longer share the same page of the address space.

18.21 In Linux, shared libraries perform many operations central to the operating system. What is the advantage of keeping this functionality out of the kernel? Are there any drawbacks? Explain your answer.
Answer:
There are a number of reasons for keeping functionality in shared libraries rather than in the kernel itself. These include:

a. **Reliability**. Kernel-mode programming is inherently higher risk than user-mode programming. If the kernel is coded correctly so that protection between processes is enforced, then an occurrence of a bug in a user-mode library is likely to affect only the currently executing process, whereas a similar bug in the kernel could conceivably bring down the entire operating system.

b. **Performance**. Keeping as much functionality as possible in user-mode shared libraries helps performance in two ways. First of all, it reduces physical memory consumption: kernel memory is non-pageable, so every kernel function is permanently resident in physical memory, but a library function can be paged in from disk on demand and does not need to be physically present all of the time. Although the library function may be resident in many processes at once, page sharing by the virtual memory system means that it is loaded at most once into physical memory.

   Second, calling a function in a loaded library is a very fast operation, but calling a kernel function through a kernel system service call is much more expensive. Entering the kernel involves changing the CPU protection domain, and once in the kernel, all of the arguments supplied by the process must be very carefully checked for correctness: the kernel cannot afford to make any assumptions about the validity of the arguments passed in, whereas a library function might reasonably do so. Both of these factors make calling a kernel function much slower than calling the same function in a library.

c. **Manageability**. Many different shared libraries can be loaded by an application. If new functionality is required in a running system, shared libraries to provide that functionality can be installed without interrupting any already running processes. Similarly, existing shared libraries can generally be upgraded without requiring any system down time. Unprivileged users can create shared libraries to be run by their own programs. All of these attributes make shared libraries generally easier to manage than kernel code.

There are, however, a few disadvantages to having code in a shared library. There are obvious examples of code that is completely unsuitable for implementation in a library, including low-level functionality such as device drivers or file systems. In general, services shared around the entire system are better implemented in the kernel if they are performance-critical, since the alternative—running the shared service in a separate process and communicating with it through interprocess communication—requires two context switches for every service requested by a process. In some cases, it may be appropriate to prototype a service in user mode but implement the final version as a kernel routine.

Security is also an issue. A shared library runs with the privileges of the process calling the library. It cannot directly access any resources inaccessible to the calling process, and the calling process has full access to all of the data structures maintained by the shared library. If the service being provided requires any privileges outside of a normal process's, or if the data managed by the library needs to be protected from normal user processes, then libraries are inappropriate and a separate server process (if performance permits) or a kernel implementation is required.

18.22 What are the benefits to a journaling filesystem such as Linux's ext3? What are the costs? Why does ext3 provide the option to journal only metadata?

Answer:

- A **journaling filesystem** such as ext3 keeps track of changes made to the filesystem in a **journal** before committing them to the filesystem. In the event of a power failure or system crash, Linux can **replay** the changes logged to the journal, preventing corruption and allowing the filesystem to come online without the need for a lengthy validity check operation.

- **Metadata-only journaling** logs only metadata and not file data operations. This yields a significant performance improvement—journaling is not cheap and metadata operations are generally but a fraction of overall operations—at the expensive of being able to recover only meta, and not file, data from the journal. This is often an acceptable trade-off as metadata-only journaling is sufficient for ensuring the filesystem is consistent after recovering from a crash or power outage.

18.23 The directory structure of a Linux operating system could include files corresponding to several different file systems, including the Linux /proc file system. How might the need to support different file-system types affect the structure of the Linux kernel?

Answer:

There are many implications to having to support different file system types within the Linux kernel. For one thing, the file system interface should be independent of the data layouts and data structures used within the file system to store file data. For another thing, it might have to provide interfaces to file systems where the file data is not static data and is not even stored on the disk; instead, the filedata could be computed every time an operation is invoked to access it, as is the case with the /proc file system. These call for a fairly general virtual interface to sit on top of the different file systems.

18.24 In what ways does the Linux setuid feature differ from the setuid feature in standard SVR4?

Answer:

Linux augments the standard setuid feature in two ways. First, it allows a program to drop and reacquire its effective uid repeatedly. In order to minimize the amount of time that a program executes with all of its privileges, a program might drop to a lower privilege level and thereby prevent the exploitation of security loopholes at the lower-level. However, when it needs to perform privileged operations, it can switch to its effective uid. Second, Linux allows a process to take on only a subset of the rights of the effective uid. For instance, an user can use a process that serves files without having control over the process in terms of being able to kill or suspend the process.

18.25 The Linux source code is freely and widely available over the Internet and from CD-ROM vendors. What are three implications of this availability for the security of the Linux system?

Answer:

The open availability of an operating system's source code has both positive and negative impacts on security, and it is probably a mistake to say that it is definitely a good thing or a bad thing.

Linux's source code is open to scrutiny by both the good guys and the bad guys. In its favor, this has resulted in the code being inspected by a large number of people who are concerned about security and who have eliminated any vulnerabilities they have found.

On the other hand is the "security through obscurity" argument, which states that attackers' jobs are made easier if they have access to the source code of the system they are trying to penetrate. By denying attackers information about a system, the hope is that it will be harder for those attackers to find and exploit any security weaknesses that may be present.

In other words, open source code implies both that security weaknesses can be found and fixed faster by the Linux community, increasing the security of the system; and that attackers can more easily find any weaknesses that do remain in Linux.

There are other implications for source code availability, however. One is that if a weakness in Linux is found and exploited, then a fix for that problem can be created and distributed very quickly. (Typically, security problems in Linux tend to have fixes available to the public within 24 hours of their discovery.) Another is that if security is a major concern to particular users, then it is possible for those users to review the source code to satisfy themselves of its level of security or to make any changes that they wish to add new security measures.

Windows 7

The Microsoft Windows 7 operating system is a 32/64-bit preemptive multitasking operating system for AMD K6/K7, Intel IA-32/IA64 and later microprocessors. The successor to Windows NT/2000, Windows 7, is also intended to replace the MS-DOS operating system. Key goals for the system are security, reliability, ease of use, Windows and POSIX application compatibility, high performance, extensibility, portability and international support. This chapter discusses the key goals for Windows 7, the layered architecture of the system that makes it so easy to use, the file system, networks, and the programming interface. Windows 7 serves as an excellent case study as an example operating system.

Exercises

19.14 Under what circumstances would one use the deferred procedure calls facility in Windows 7?

Answer:

Deferred procedure calls are used to postpone interrupt processing in situations where the processing of device interrupts can be broken into a critical portion that is used to unblock the device and a non-critical portion that can be scheduled later at a lower priority. The non-critical section of code is scheduled for later execution by queuing a deferred procedure call.

19.15 What is a handle, and how does a process obtain a handle?

Answer:

User-mode code can access kernel-mode objects by using a reference value called a handle. An object handle is thus an identifier (unique to a process) that allows access and manipulation of a system resource. When a user-mode process wants to use an object it calls the object manager's open method. A reference to the object is inserted in the process's object table and a handle is returned. Processes can obtain handles by creating an object, opening an existing object, receiving a duplicated handle from another process, or inheriting a handle from a parent process.

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

Answer:

The VM manager uses a page-based management scheme. Pages of data allocated to a process that are not in physical memory are either stored in paging files on disk or mapped to a regular file on a local or remote file system. To improve performance of this scheme, a privileged process is allowed to lock selected pages in physical memory preventing those pages from being paged out. Furthermore, since when a page is used, adjacent pages will likely be used in the near future, adjacent pages are prefetched to reduce the total number of page faults.

19.17 Describe a useful application of the no-access page facility provided in Windows 7.

Answer:

When a process accesses a no-access page, an exception is raised. This feature is used to check whether a faulty program accesses beyond the end of an array. The array needs to be allocated in a manner such that it appears at the end of a page, so that buffer overruns would cause exceptions.

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

Answer:

Data is communicated using one of the following three facilities: 1) messages are simply copied from one process to the other, 2) a shared memory segment is created and messages simply contain a pointer into the shared memory segment, thereby avoiding copies between processes, 3) a process directly writes into the other process's virtual space.

19.19 What manages caching in Windows 7? How is the cache managed?

Answer:

In contrast to other operating systems where caching is done by the file system, Windows 7 provides a centralized cache manager which works closely with the VM manager to provide caching services for all components under control of the I/O manager. The size of the cache changes dynamically depending upon the free memory available in the system. The cache manager maps files into the upper half of the system cache address space. This cache is divided into blocks that can each hold a memory-mapped region of a file.

19.20 How does the NTFS directory structure differ from the directory structure used in UNIX operating systems?

Answer:

The NTFS namespace is organized as a hierarchy of directories where each directory uses a B+ tree data structure to store an index of the filenames in that directory. The index root of a directory contains the top level of the B+ tree. Each entry in the directory contains the name and file reference of the file as well as the update timestamp and file size. The UNIX operating system simply stores a table of entries mapping names to i-node numbers in a directory file. Lookups and updates require a linear scan of the directory structure in UNIX systems.

19.21 What is a process, and how is it managed in Windows 7?

Answer:

A process is an executing instance of an application containing one or more threads. Threads are the units of code that are scheduled by the operating system. A process is started when some other process calls the CreateProcess routine, which loads any dynamic link libraries used by the process, resulting in a primary thread. Additional threads can also be created. Each thread is created with its own stack with a wrapper function providing thread synchronization.

19.22 What is the fiber abstraction provided by Windows 7? How does it differ from the threads abstraction?

Answer:

A fiber is a sequential stream of execution within a process. A process can have multiple fibers in it, but unlike threads, only one fiber at a time is permitted to execute. The fiber mechanism is used to support legacy applications written for a fiber-execution model.

19.23 How does user-mode scheduling (UMS) in Windows 7 differ from fibers? What are some trade-offs between fibers and UMS?

Answer:

Fibers are only UTs, and the kernel has no knowledge of their existence. They do not have their own TEBs and thus cannot reliably run Windows 7 APIs. Because a UT shares the KT of the thread it executes on, it must be careful not to change the state of the KT, such as by using impersonation or canceling I/O. Fibers lose control of the CPU whenever the borrowed KT blocks in the kernel. With UMS, control of the CPU is always returned to the user-mode scheduler.

19.24 UMS considers a thread to have two parts, a UT and a KT. How might it be useful to allow UTs to continue executing in parallel with their KTs?

Answer:

If a UT could continue executing even though its corresponding KT was running in the kernel, it would allow the system services performed by KTs to be asynchronous—as I/O already is in Windows 7. However, the programming language or run-time would have to provide a way of synchronizing with the results of the system service being performed by each asynchronous KT.

19.25 What is the performance trade-off of allowing KTs and UTs to execute on different processors?

Answer:

If KTs and UTs ran on different processors, the application would have more concurrency and better cache locality—since the KT would not be poisoning the cache used by the UT. However, there would be more latency, as the CPU that ran the KT might be busy with other work when the service request arrived. In addition, there would be higher overheads due to queuing and synchronization costs.

19.26 Why does the self-map occupy large amounts of virtual address space but no additional virtual memory?

Answer:

The virtual memory pages used for the page table have to be allocated whether or not the pages are mapped into the kernel virtual address space. The only cost is the loss of use of one of the PDE entries in the highest-level page directory.

19.27 How does the self-map make it easy for the VM manager to move the page table pages to and from disk? Where are the page-table pages kept on disk?

Answer:

The self-map places the page-table pages into the kernel's virtual address space. The pages are part of kernel virtual memory, so the VM manager can page them in and out of memory just as it does other virtual memory. Because the page-table pages are not backed by a memory-mapped file, they are kept in the page file when not in memory.

19.28 When a Windows 7 system hibernates, the system is powered off. Suppose you changed the CPU or the amount of RAM on a hibernating system. Do you think that would work? Why or why not?

Answer:

It would likely cause the system to crash. Information about the CPU and RAM (as well as many other specifics about the hardware) are captured by the kernel when it boots. When the system resumes from hibernation, the data structures that describe this information will be inconsistent with the changed hardware.

19.29 Give an example showing how the use of a suspend count is helpful in suspending and resuming threads in Windows 7.

Answer:

Suppose an operation suspends a thread, examines some state while the thread is suspended, and then resumes the thread. If two different threads attempt the operation on the same thread, the use of suspend counts will keep the target thread from prematurely resuming.

Influential Operating Systems

CHAPTER 20

Now that you understand the fundamental concepts of operating systems (CPU scheduling, memory management, processes, and so on), we are in a position to examine how these concepts have been applied in several older and highly influential operating systems. Some of them (such as the XDS-940 and the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. The order of presentation highlights the similarities and differences of the systems; it is not strictly chronological or ordered by importance. The serious student of operating systems should be familiar with all these systems.

Exercises

20.1 Discuss what considerations the computer operator took into account in deciding on the sequences in which programs would be run on early computer systems that were manually operated.

Answer:
Jobs with similar needs are batched together and run together to reduce set-up time. For instance, jobs that require the same compiler because they were written in the same language are scheduled together so that the compiler is loaded only once and used on both programs.

20.2 What optimizations were used to minimize the discrepancy between CPU and I/O speeds on early computer systems?

Answer:
An optimization used to minimize the discrepancy between CPU and I/O speeds is spooling. Spooling overlaps the I/O of one job with the computation of other jobs. The spooler for instance could be reading the input of one job while printing the output of a different job or while executing another job.

20.3 Consider the page replacement algorithm used by Atlas. In what ways is it different from the clock algorithm discussed in Section 9.4.5.2?

Answer:
The page replacement algorithm used in Atlas is very different from the clock algorithm discussed in earlier chapters. The Atlas system keeps track of whether a page was accessed in each period of 1024 instructions

for the last 32 periods. Let t1 be the time since the most recent reference to a page, while t2 is the interval between the last two references of a page. The paging system then discards any page that has t1 > t2 + 1. If it cannot find any such page, it discards the page with the largest t2 - t1. This algorithm assumes that programs access memory in loops and the idea is to retain pages even if it has not been accessed for a long time if there has been a history of accessing the page regularly albeit at long intervals. The clock algorithm, on the other hand, is an approximate version of the least recently used algorithm and therefore discards the least recently used page without taking into account that some of the pages might be infrequently but repeatedly accessed.

20.4 Consider the multilevel feedback queue used by CTSS and MULTICS. Suppose a program consistently uses seven time units every time it is scheduled before it performs an I/O operation and blocks. How many time units are allocated to this program when it is scheduled for execution at different points in time?

Answer:

Assume that the process is initially scheduled for one time unit. Since the process does not finish by the end of the time quantum, it is moved to a lower level queue and its time quantum is raised to two time units. This process continues till it is moved to a level 4 queue with a time quantum of 8 time units. In certain multilevel systems, when the process executes next and does not use its full time quantum, the process might be moved back to a level 3 queue.

20.5 What are the implications of supporting BSD functionality in user-mode servers within the Mach operating system?

Answer:

Mach operating system supports the BSD functionality in user mode servers. When the user process makes a BSD call, it traps into kernel mode and the kernel copies the arguments to the user level server. A context switch is then made and the user level performs the requested operation and computes the results which are then copied back to the kernel space. Another context switch takes place to the original process which is in kernel mode and the process eventually transitions from kernel mode to user mode along with the results of the BSD call. Therefore, in order to perform the BSD call, there are two kernel crossings and two process switches thereby resulting in a large overhead. This is significantly higher than the cost if the BSD functionality is supported within the kernel.

20.6 What conclusions can be drawn about the evolution of operating systems? What causes some operating systems to gain in popularity and others to fade?

Answer:

Operating systems that have made advances in operating system technology —that, is advances to memory management or interprocess communication — have typically been the types of systems that are both popular and have existed for a period of time. UNIX is a classic example of a type of system that has made significant technological advances and has lasted for more than 30 years. Further evidence of this are the types of systems that have evolved from UNIX and have gained in popularity on their own. Linux is perhaps the most notable example. The types of operating systems that have faded from view are typically systems that are either too specific in purpose or lack in performance. There is much more motivation to replace such systems rather than continuing to advance them.