CHAPTER 1

Introduction

Practice Exercises

1.1 What are the three main purposes of an operating system?

Answer:
The three main purposes are:

- To provide an environment for a computer user to execute programs on computer hardware in a convenient and efficient manner.
- To allocate the separate resources of the computer as needed to perform the required tasks. The allocation process should be as fair and efficient as possible.
- As a control program, it serves two major functions: (1) supervision of the execution of user programs to prevent errors and improper use of the computer, and (2) management of the operation and control of I/O devices.

1.2 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?

Answer:
Single-user systems should maximize use of the system for the user. A GUI might "waste" CPU cycles, but it optimizes the user's interaction with the system.

1.3 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

Answer:
The main difficulty is keeping the operating system within the fixed time constraints of a real-time system. If the system does not complete a task in a certain time frame, it may cause a breakdown of the entire system. Therefore, when writing an operating system for a real-time system, the

writer must be sure that his scheduling schemes don't allow response time to exceed the time constraint.

1.4 Keeping in mind the various definitions of ***operating system***, consider whether the operating system should include applications such as web browsers and mail programs. Argue both that it should and that it should not, and support your answers.

Answer:
An argument in favor of including popular applications in the operating system is that if the application is embedded within the operating system, it is likely to be better able to take advantage of features in the kernel and therefore have performance advantages over an application that runs outside of the kernel. Arguments against embedding applications within the operating system typically dominate, however: (1) the applications are applications—not part of an operating system, (2) any performance benefits of running within the kernel are offset by security vulnerabilities, and (3) inclusion of applications leads to a bloated operating system.

1.5 How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security)?

Answer:
The distinction between kernel mode and user mode provides a rudimentary form of protection in the following manner. Certain instructions can be executed only when the CPU is in kernel mode. Similarly, hardware devices can be accessed only when the program is in kernel mode, and interrupts can be enabled or disabled only when the CPU is in kernel mode. Consequently, the CPU has very limited capability when executing in user mode, thereby enforcing protection of critical resources.

1.6 Which of the following instructions should be privileged?

a. Set value of timer.
b. Read the clock.
c. Clear memory.
d. Issue a trap instruction.
e. Turn off interrupts.
f. Modify entries in device-status table.
g. Switch from user to kernel mode.
h. Access I/O device.

Answer:
The following operations need to be privileged: set value of timer, clear memory, turn off interrupts, modify entries in device-status table, access I/O device. The rest can be performed in user mode.

1.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.

Answer:
The data required by the operating system (passwords, access controls, accounting information, and so on) would have to be stored in or passed through unprotected memory and thus be accessible to unauthorized users.

1.8 Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?

Answer:
Although most systems only distinguish between user and kernel modes, some CPUs have supported multiple modes. Multiple modes could be used to provide a finer-grained security policy. For example, rather than distinguishing between just user and kernel mode, you could distinguish between different types of user mode. Perhaps users belonging to the same group could execute each other's code. The machine would go into a specified mode when one of these users was running code. When the machine was in this mode, a member of the group could run code belonging to anyone else in the group.

Another possibility would be to provide different distinctions within kernel code. For example, a specific mode could allow USB device drivers to run. This would mean that USB devices could be serviced without having to switch to kernel mode, thereby essentially allowing USB device drivers to run in a quasi-user/kernel mode.

1.9 Timers could be used to compute the current time. Provide a short description of how this could be accomplished.

Answer:
A program could use the following approach to compute the current time using timer interrupts. The program could set a timer for some time in the future and go to sleep. When awakened by the interrupt, it could update its local state, which it uses to keep track of the number of interrupts it has received thus far. It could then repeat this process of continually setting timer interrupts and updating its local state when the interrupts are actually raised.

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

Answer:
Caches are useful when two or more components need to exchange data, and the components perform transfers at differing speeds. Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device. The data in the cache must be kept consistent with the data in the components. If a component has

a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems, where more than one process may be accessing a datum. A component may be eliminated by an equal-sized cache, but only if: (a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and (b) the cache is affordable, because faster storage tends to be more expensive.

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

Answer:

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

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

Operating-System Structures

Practice Exercises

2.1 What is the purpose of system calls?

Answer:
System calls allow user-level processes to request services of the operating system.

2.2 What is the purpose of the command interpreter? Why is it usually separate from the kernel?

Answer:
It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls. It is usually not part of the kernel because the command interpreter is subject to changes.

2.3 What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?

Answer: A `fork()` system call and an `exec()` system call need to be performed to start a new process. The `fork()` call clones the currently executing process, while the `exec()` call overlays a new process based on a different executable over the calling process.

2.4 What is the purpose of system programs?

Answer:
System programs can be thought of as bundles of useful system calls. They provide basic functionality to users so that users do not need to write their own programs to solve common problems.

2.5 What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?

Answer:
As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather

than touching all sections. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer. The primary disadvantage to the layered approach is th apoor performance due to the overhead of traversing through the different layers to obtain a service provided by the operating system.

2.6 List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.

Answer:
The five services are:

a. **Program execution**. The operating system loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate CPU time.

b. **I/O operations**. It is necessary to communicate with disks, tapes, and other devices at a very low level. The user need only specify the device and the operation to perform on it, and the system converts that request into device- or controller-specific commands. User-level programs cannot be trusted to access only devices they should have access to and to access them only when they are otherwise unused.

c. **File-system manipulation**. There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must also be checked to assure proper file access. User programs could neither ensure adherence to protection methods nor be trusted to allocate only free blocks and deallocate blocks on file deletion.

d. **Communications**. Message passing between systems requires messages to be turned into packets of information, sent to the network controller, transmitted across a communications medium, and reassembled by the destination system. Packet ordering and data correction must take place. Again, user programs might not coordinate access to the network device, or they might receive packets destined for other processes.

e. **Error detection**. Error detection occurs at both the hardware and software levels. At the hardware level, all data transfers must be inspected to ensure that data have not been corrupted in transit. All data on media must be checked to be sure they have not changed since they were written to the media. At the software level, media must be checked for data consistency—for instance, whether the number of allocated and unallocated blocks of storage match the total number on the device. There, errors are frequently process-independent (for instance, the corruption of data on a disk), so there must be a global program (the operating system) that handles

all types of errors. Also, when errors are processed by the operating system, processes need not contain code to catch and correct all the errors possible on a system.

2.7 Why do some systems store the operating system in firmware, while others store it on disk?

Answer:
For certain devices, such as embedded systems, a disk with a file system may be not be available for the device. In this situation, the operating system must be stored in firmware.

2.8 How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

Answer:
Consider a system that would like to run both Windows and three different distributions of Linux (for example, RedHat, Debian, and Ubuntu). Each operating system will be stored on disk. During system boot, a special program (which we will call the **boot manager**) will determine which operating system to boot into. This means that rather than initially booting to an operating system, the boot manager will first run during system startup. It is this boot manager that is responsible for determining which system to boot into. Typically, boot managers must be stored at certain locations on the hard disk to be recognized during system startup. Boot managers often provide the user with a selection of systems to boot into; boot managers are also typically designed to boot into a default operating system if no choice is selected by the user.

CHAPTER 3

Processes

Practice Exercises

3.1 Using the program shown in Figure 3.30, explain what the output will be at LINE A.

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

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Figure 3.30 What output will be at Line A?

Answer:
The result is still 5, as the child updates its copy of value. When control returns to the parent, its value remains at 5.

3.2 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

Answer:
Eight processes are created.

3.3 Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

Answer:

a. The CPU scheduler must be aware of the different concurrent processes and must choose an appropriate algorithm that schedules the concurrent processes.

b. Concurrent processes may need to communicate with one another, and the operating system must therefore develop one or more methods for providing interprocess communication.

c. Because mobile devices often have limited memory, a process that manages memory poorly will have an overall negative impact on other concurrent processes. The operating system must therefore manage memory to support multiple concurrent processes.

3.4 Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

Answer:
The CPU current-register-set pointer is changed to point to the set containing the new context, which takes very little time. If the context is in memory, one of the contexts in a register set must be chosen and be moved to memory, and the new context must be loaded from memory into the set. This process takes a little more time than on systems with one set of registers, depending on how a replacement victim is selected.

3.5 When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments

Answer:
Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

3.6 Consider the "exactly once"semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly

even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether "exactly once" is still preserved.

Answer:

The "exactly once" semantics ensure that a remore procedure will be executed exactly once and only once. The general algorithm for ensuring this combines an acknowledgment (ACK) scheme combined with timestamps (or some other incremental counter that allows the server to distinguish between duplicate messages).

The general strategy is for the client to send the RPC to the server along with a timestamp. The client will also start a timeout clock. The client will then wait for one of two occurrences: (1) it will receive an ACK from the server indicating that the remote procedure was performed, or (2) it will time out. If the client times out, it assumes the server was unable to perform the remote procedure, so the client invokes the RPC a second time, sending a later timestamp. The client may not receive the ACK for one of two reasons: (1) the original RPC was never received by the server, or (2) the RPC was correctly received—and performed—by the server but the ACK was lost. In situation (1), the use of ACKs allows the server ultimately to receive and perform the RPC. In situation (2), the server will receive a duplicate RPC, and it will use the timestamp to identify it as a duplicate so as not to perform the RPC a second time. It is important to note that the server must send a second ACK back to the client to inform the client the RPC has been performed.

3.7 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the "exactly once" semantic for execution of RPCs?

Answer:

The server should keep track in stable storage (such as a disk log) of information regarding what RPC operations were received, whether they were successfully performed, and the results associated with the operations. When a server crash takes place and an RPC message is received, the server can check whether the RPC has been previously performed and therefore guarantee "exactly once" semantics for the execution of RPCs.

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.31 How many processes are created?

Threads & Concurrency

Practice Exercises

4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

Answer:

a. A web server that services each request in a separate thread

b. A parallelized application such as matrix multiplication where various parts of the matrix can be worked on in parallel

c. An interactive GUI program such as a debugger where one thread is used to monitor user input, another thread represents the running application, and a third thread monitors performance

4.2 Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

Answer:

a. With two processing cores we get a speedup of 1.42 times.

b. With four processing cores, we get a speedup of 1.82 times.

4.3 Does the multithreaded web server described in Section 4.1 exhibit task or data parallelism?

Answer:
Data parallelism. Each thread is performing the same task, but on different data.

4.4 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

Answer:

a. User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads.

b. On systems using either many-to-one or many-to-many model mapping, user threads are scheduled by the thread library, and the kernel schedules kernel threads.

c. Kernel threads need not be associated with a process, whereas every user thread belongs to a process. Kernel threads are generally more expensive to maintain than user threads, as they must be represented with a kernel data structure.

4.5 Describe the actions taken by a kernel to context-switch between kernel-level threads.

Answer:
Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

4.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

Answer:
Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, a list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user thread or a kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

4.7 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

Answer:
Yes. Timing is crucial to real-time applications. If a thread is marked as real-time but is not bound to an LWP, the thread may have to wait to be attached to an LWP before running. Consider a situation in which a real-time thread is running (is attached to an LWP) and then proceeds to block (must perform I/O, has been preempted by a higher-priority real-time thread, is waiting for a mutual exclusion lock, etc.). While the real-time thread is blocked, the LWP it was attached to is assigned to another thread. When the real-time thread has been scheduled to run again, it must first wait to be attached to an LWP. By binding an LWP to a real-time thread, you are ensuring that the thread will be able to run with minimal delay once it is scheduled.

CPU Scheduling

Practice Exercises

5.1 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n.

Answer:
$n!$ (n factorial = $n \times n-1 \times n-2 \times \ldots \times 2 \times 1$).

5.2 Explain the difference between preemptive and nonpreemptive scheduling.

Answer:
Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

5.3 Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know

that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as ***future-knowledge scheduling***.

Answer:

a. 10.53

b. 9.53

c. 6.86

Remember that turnaround time is finishing time minus arrival time, so you have to subtract the arrival times to compute the turnaround times. FCFS is 11 if you forget to subtract arrival time.

5.4 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).

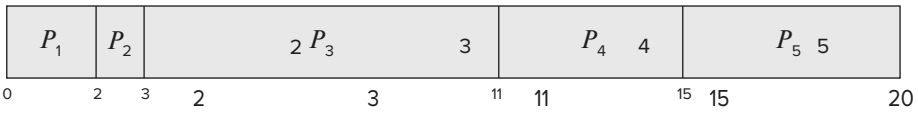
b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

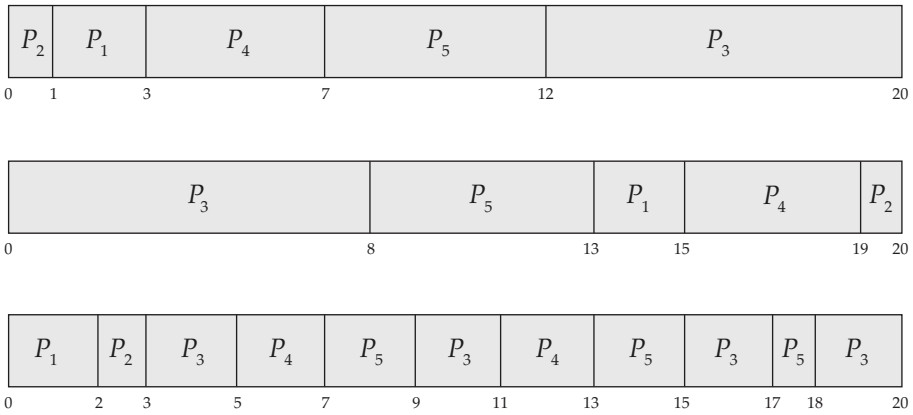
c. What is the waiting time of each process for each of these scheduling algorithms?

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

Answer:

a. The four Gantt charts:





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. SJF has the shortest wait time.

5.5 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm.

Process	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

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed above, 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.

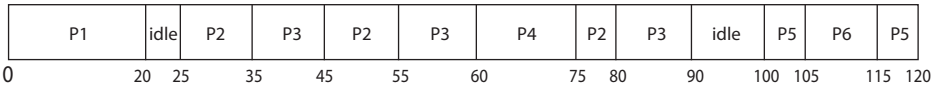
b. What is the turnaround time for each process?

c. What is the waiting time for each process?

d. What is the CPU utilization rate?

Answer:

a. The Gantt chart:



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. P1: 0, p2: 40, P3: 35, P4: 0, P5: 10, P6: 0

d. 105/120 = 87.5 percent.

5.6 What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?

Answer:
Processes that need more frequent servicing—for instance, interactive processes such as editors—can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing and thus making more efficient use of the computer.

5.7 Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

a. Priority and SJF

b. Multilevel feedback queues and FCFS

c. Priority and FCFS

d. RR and SJF

Answer:

a. The shortest job has the highest priority.

b. The lowest level of MLFQ is FCFS.

c. FCFS gives the highest priority to the job that has been in existence the longest.

d. None.

5.8 Suppose that a CPU scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

Answer:
It will favor the I/O-bound programs because of the relatively short CPU bursts requested by them; however, the CPU-bound programs will not starve, because the I/O-bound programs will relinquish the CPU relatively often to do their I/O.

5.9 Distinguish between PCS and SCS scheduling.

Answer:
PCS scheduling is local to the process. It is how the thread library schedules threads onto available LWPs. SCS scheduling is used when the operating system schedules kernel threads. On systems using either the many-to-one or the many-to-many model, the two scheduling models are fundamentally different. On systems using the one-to-one model, PCS and SCS are the same.

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

Priority = (recent CPU usage / 2) + base

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

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

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

Synchronization Tools

CHAPTER 6

Practice Exercises

6.1 In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

Answer:
The system clock is updated at every clock interrupt. If interrupts were disabled—particularly for a long period of time—the system clock could easily lose the correct time. The system clock is also used for scheduling purposes. For example, the time quantum for a process is expressed as a number of clock ticks. At every clock interrupt, the scheduler determines if the time quantum for the currently running process has expired. If clock interrupts were disabled, the scheduler could not accurately assign time quanta. This effect can be minimized by disabling clock interrupts for only very short periods.

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

Answer:
Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. One strategy to avoid busy waiting temporarily puts the waiting process to sleep and awakens it when the appropriate program state is reached, but this solution incurs the overhead associated with putting the process to sleep and later waking it up.

6.3 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

Answer:
Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the

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

6.4 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

Answer:
A `wait()` operation atomically decrements the value associated with a semaphore. If two `wait()` operations are executed on a semaphore when its value is 1 and the operations are not performed atomically, then both operations might decrement the semaphore value, thereby violating mutual exclusion.

6.5 Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.

Answer:
The n processes share a semaphore, `mutex`, initialized to 1. Each process P_i is organized as follows:

```
do {
   wait(mutex);

      /* critical section */

   signal(mutex);

      /* remainder section */
} while (true);
```

6.6 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Answer:
Assume that the balance in the account is $250.00 and that 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 the 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 the 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.

Synchronization Examples

Practice Exercises

7.1 Explain why Windows and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, and condition variables. In each case, explain why the mechanism is needed.

Answer:
These operating systems provide different locking mechanisms depending on the application developers' needs. Spinlocks are useful for multiprocessor systems where a thread can run in a busy loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue. Mutexes are useful for locking resources. Solaris 2 uses adaptive mutexes, meaning that the mutex is implemented with a spinlock on multiprocessor machines. Semaphores and condition variables are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration.

7.2 Windows provides a lightweight synchronization tool called **slim reader–writer** locks. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.

Answer:
Simplicity. If reader–writer locks provide fairness or favor readers or writers, they involve more overhead. Providing such a simple synchronization mechanism makes access to the lock fast. Use of this lock may be most appropriate for situations where reader–writer locks are needed, but quickly acquiring and releasing them is similarly important.

7.3 Describe what changes would be necessary to the producer and consumer processes in Figure 7.1 and Figure 7.2 so that a mutex lock could be used instead of a binary semaphore.

Answer:

The calls to `wait(mutex)` and `signal(mutex)` need to be replaced so that they are now calls to the API for a mutex lock, such as `acquire(mutex)` and `release()` mutex.

7.4 Describe how deadlock is possible with the dining-philosophers problem.

Answer:
If all philosophers simultaneously pick up their left forks, when they turn to pick up their right forks they will realize they are unavailable, and will block while waiting for it to become available. This blocking while waiting for a resource to become available is a deadlocked situation.

7.5 Explain the difference between signaled and non-signaled states with Windows dispatcher objects.

Answer:
An object that is in the signaled state is available, and a thread will not block when it tries to acquire it. When the lock is acquired, it is in the non-signaled state. When the lock is released, it transitions back to the signaled state.

7.6 Assume `val` is an atomic integer in a Linux system. What is the value of `val` after the following operations have been completed?

```
atomic_set(&val,10);
atomic_sub(8,&val);
atomic_inc(&val);
atomic_inc(&val);
atomic_add(6,&val);
atomic_sub(3,&val);
```

Answer:
The final value of `val` is 10 - 8 + 1 + 1 + 6 - 3 = 7

CHAPTER 8

Deadlocks

Practice Exercises

8.1 List three examples of deadlocks that are not related to a computer-system environment.

Answer:

- Two cars crossing a single-lane bridge from opposite directions.
- A person going down a ladder while another person is climbing up the ladder.
- Two trains traveling toward each other on the same track.

8.2 Suppose that a system is in an unsafe state. Show that it is possible for the threads to complete their execution without entering a deadlocked state.

Answer:
An unsafe state may not necessarily lead to deadlock, it just means that we cannot guarantee that deadlock will not occur. Thus, it is possible that a system in an unsafe state may still allow all processes to complete without deadlock occurring. Consider the situation where a system has twelve resources allocated among processes P_0, P_1, and P_2. The resources are allocated according to the following policy:

	Max	Current	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

Currently, there are two resources available. This system is in an unsafe state. Process P_1 could complete, thereby freeing a total of four resources, but we cannot guarantee that processes P_0 and P_2 can complete. However, it is possible that a process may release resources before requesting any further resources. For example, process P_2 could release a resource, thereby increasing the total number of resources to five. This allows pro-

cess P_0 to complete, which would free a total of nine resources, thereby allowing process P_2 to complete as well.

8.3 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
T_0	0 0 1 2	0 0 1 2	1 5 2 0
T_1	1 0 0 0	1 7 5 0	
T_2	1 3 5 4	2 3 5 6	
T_3	0 6 3 2	0 6 5 2	
T_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix ***Need***?

b. Is the system in a safe state?

c. If a request from thread T_1 arrives for (0,4,2,0), can the request be granted immediately?

Answer:

a. The values of ***Need*** for processes P_0 through P_4, respectively, are (0, 0, 0, 0), (0, 7, 5, 0), (1, 0, 0, 2), (0, 0, 2, 0), and (0, 6, 4, 2).

b. The system is in a safe state. With ***Available*** equal to (1, 5, 2, 0), either process P_0 or P_3 could run. Once process P_3 runs, it releases its resources, which allows all other existing processes to run.

c. The request can be granted immediately. The value of ***Available*** is then (1, 1, 0, 0). One ordering of processes that can finish is P_0, P_2, P_3, P_1, and P_4.

8.4 A possible method for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects $A \cdots E$, deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent deadlock by adding a sixth object F. Whenever a thread wants to acquire the synchronization lock for any object $A \cdots E$, it must first acquire the lock for object F. This solution is known as **containment**: the locks for objects $A \cdots E$ are contained within the lock for object F. Compare this scheme with the circular-wait scheme of Section 8.5.4.

Answer:

This is probably not a good solution because it yields too large a scope. It is better to define a locking policy with as narrow a scope as possible. The circular wait approach is a reasonable approach to avoiding deadlock, and does not increase the scope of holding a lock.

8.5 Prove that the safety algorithm presented in Section 8.6.3 requires an order of $m \times n^2$ operations.

Answer:

The figure below provides Java code that implements the safety algorithm of the banker's algorithm (the complete implementation of the banker's algorithm is available with the source-code download for this text).

```
for (int i = 0; i < n; i++) {
   // first find a thread that can finish
   for (int j = 0; j < n; j++) {
      if (!finish[j]) {
         boolean temp = true;
         for (int k = 0; k < m; k++) {
            if (need[j][k] > work[k])
               temp = false;
         }

         if (temp) { // if this thread can finish
            finish[j] = true;
            for (int x = 0; x < m; x++)
               work[x] += work[j][x];
         }
      }
   }
}
```

As can be seen, the nested outer loops—both of which loop through n times—provide the n^2 performance. Within these outer loops are two sequential inner loops that loop m times. The Big O of this algorithm is therefore $O(m \times n^2)$.

8.6 Consider a computer system that runs 5,000 jobs per month and has no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about ten jobs per deadlock. Each job is worth about two dollars (in CPU time), and the jobs terminated tend to be about half done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase of about 10 percent in the average execution time per job. Since the machine currently has 30 percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

a. What are the arguments for installing the deadlock-avoidance algorithm?

b. What are the arguments against installing the deadlock-avoidance algorithm?

Answer:

An argument for installing deadlock avoidance in the system is that we could ensure that deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run.
An argument against installing deadlock-avoidance software is that deadlocks occur infrequently, and they cost little when they do occur.

8.7 Can a system detect that some of its threads are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

Answer:
Starvation is a difficult topic to define, as it may mean different things for different systems. For the purposes of this question, we will define starvation as the situation in which a process must wait beyond a reasonable period of time—perhaps indefinitely—before receiving a requested resource. One way of detecting starvation would be to first identify a period of time—T—that is considered unreasonable. When a process requests a resource, a timer is started. If the elapsed time exceeds T, then the process is considered to be starved.

One strategy for dealing with starvation would be to adopt a policy whereby resources are assigned only to the process that has been waiting the longest. For example, if process P_a has been waiting longer for resource X than process P_b, the request from process P_b would be deferred until process P_a's request has been satisfied.

Another strategy would be less strict. In this scenario, a resource might be granted to a process that had waited less than another process, providing that the other process was not starving. However, if another process was considered to be starving, its request would be satisfied first.

8.8 Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any threads that are blocked waiting for resources. If a blocked thread has the desired resources, then these resources are taken away from it and are given to the requesting thread. The vector of resources for which the blocked thread is waiting is increased to include the resources that were taken away.

For example, a system has three resource types, and the vector ***Available*** is initialized to (4,2,2). If thread T_0 asks for (2,2,1), it gets them. If T_1 asks for (1,0,1), it gets them. Then, if T_0 asks for (0,0,1), it is blocked (resource not available). If T_2 now asks for (2,0,0), it gets the available one (1,0,0), as well as one that was allocated to T_0 (since T_0 is blocked). T_0's ***Allocation*** vector goes down to (1,2,1), and its ***Need*** vector goes up to (1,0,1).

a. Can deadlock occur? If you answer "yes," give an example. If you answer "no," specify which necessary condition cannot occur.

b. Can indefinite blocking occur? Explain your answer.

Answer:

a. Deadlock cannot occur, because preemption exists.

b. Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process *C*.

8.9 Consider the following snapshot of a system:

	Allocation	***Max***
	A B C D	*A B C D*
T_0	3 0 1 4	5 1 1 7
T_1	2 2 1 0	3 2 1 1
T_2	3 1 2 1	3 3 2 1
T_3	0 5 1 0	4 6 1 2
T_4	4 2 1 2	6 3 2 5

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

a. ***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.

8.10 Suppose that you have coded the deadlock-avoidance safety algorithm that determines if a system is in a safe state or not, and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining $\boldsymbol{Max}_i = \boldsymbol{Waiting}_i + \boldsymbol{Allocation}_i$, where $\boldsymbol{Waiting}_i$ is a vector specifying the resources for which thread i is waiting and $\boldsymbol{Allocation}_i$ is as defined in Section 8.6? Explain your answer.

Answer:
Yes. The *Max* vector represents the maximum request a process may make. When calculating the safety algorithm, we use the *Need* matrix, which represents *Max* — *Allocation*. Another way to think of this is *Max* = *Need* + *Allocation*. According to the question, the *Waiting* matrix fulfills a role similar to the *Need* matrix; therefore, *Max* = *Waiting* + *Allocation*.

8.11 Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

Answer:
No. This follows directly from the hold-and-wait condition.

Main Memory

Practice Exercises

9.1 Name two differences between logical and physical addresses.

Answer:
A logical address does not refer to an actual physical address; rather, it refers to an abstract address in an abstract address space. A physical address refers to an actual physical address in memory. A logical address is generated by the CPU and is translated into a physical address by the memory management unit(MMU). Therefore, physical addresses are generated by the MMU.

9.2 Why are page sizes always powers of 2?

Answer:
Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

9.3 Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base–limit register pairs are provided: one for instructions and one for data. The instruction base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

Answer:
The major advantage of this scheme is that it is an effective mechanism for code and data sharing. For example, only one copy of an editor or a compiler needs to be kept in memory, and this code can be shared by all processes needing access to the editor or compiler code. Another advantage is protection of code against erroneous modification. The only

disadvantage is that the code and data must be separated, which is usually adhered to in a compiler-generated code.

9.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

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

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

Answer:

a. Logical address: 16 bits

b. Physical address: 15 bits

9.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?

Answer:

By allowing two entries in a page table to point to the same page frame in memory, users can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers, and database systems. "Copying" large amounts of memory could be effected by having different page tables point to the same memory location.

However, sharing of nonreentrant code or data means that any user having access to the code can modify it, and these modifications would be reflected in the other user's "copy."

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

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

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

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

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

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

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the BTV operating system?

Answer:
Conventional, single-level page table will have $2^{10} = 1024$ entries. Inverted page table will have $2^5 = 32$ entries. The maximum amount of physical memory is $2^{16} = 65536$ (or 64 KB.)

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

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

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

Answer:

a. 12 + 8 = 20 bits.

b. 12 + 6 = 18 bits.

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

a. A conventional, single-level page table

b. An inverted page table

Answer:

a. 2^{20} entries.

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

Virtual Memory

Practice Exercises

10.1 Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

Answer:
A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated, and the instruction is restarted.

10.2 Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p, and n distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:

a. What is a lower bound on the number of page faults?

b. What is an upper bound on the number of page faults?

Answer:

a. n

b. p

10.3 Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rate. Separate those algorithms that suffer from Belady's anomaly from those that do not.

a. LRU replacement

b. FIFO replacement

c. Optimal replacement

d. Second-chance replacement

Answer:

Rank	Algorithm	Suffer from Belady's anomaly
1	Optimal	no
2	LRU	no
3	Second-chance	yes
4	FIFO	yes

10.4 An operating system supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfers 1 million words per second. The following statistical measurements were obtained from the system:

- One percent of all instructions executed accessed a page other than the current page.
- Of the instructions that accessed another page, 80 percent accessed a page already in memory.
- When a new page was required, the replaced page was modified 50 percent of the time.

Calculate the effective instruction time on this system, assuming that the system is running one process only and that the processor is idle during drum transfers.

Answer:

$$\begin{aligned}\text{effective access time} &= 0.99 \times (1\ \mu\text{sec} + 0.008 \times (2\ \mu\text{sec}) \\ &\quad + 0.002 \times (10{,}000\ \mu\text{sec} + 1{,}000\ \mu\text{sec}) \\ &\quad + 0.001 \times (10{,}000\ \mu\text{sec} + 1{,}000\ \mu\text{sec}) \\ &= (0.99 + 0.016 + 22.0 + 11.0)\ \mu\text{sec} \\ &= 34.0\ \mu\text{sec}\end{aligned}$$

10.5 Consider the page table for a system with 12-bit virtual and physical addresses and 256-byte pages.

Page	Page Frame
0	–
1	2
2	C
3	A
4	–
5	4
6	3
7	–
8	B
9	0

The list of free page frames is *D*, *E*, *F* (that is, *D* is at the head of the list, *E* is second, and *F* is last). A dash for a page frame indicates that the page is not in memory.

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal.

- 9EF
- 111
- 700
- 0FF

Answer:

- 9*EF* → 0*EF*
- 111 → 211
- 700 → *D*00
- 0*FF* → *EFF*

10.6 Discuss the hardware functions required to support demand paging.

Answer:

For every memory-access operation, the page table must be consulted to check whether the corresponding page is resident and whether the program has read or write privileges for accessing the page. These checks must be performed in hardware. A TLB could serve as a cache and improve the performance of the lookup operation.

10.7 Consider the two-dimensional array `A`:

```
int A[][] = new int[100][100];
```

where `A[0][0]` is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops? Use LRU replacement, and assume

that page frame 1 contains the process and the other two are initially empty.

a.
```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```

b.
```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

Answer:

a. 5,000

b. 50

10.8 Consider the following page reference string:

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

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

Answer:

Number of frames	LRU	FIFO	Optimal
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7

10.9 Consider the following page reference string:

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

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

- LRU replacement
- FIFO replacement

- Optimal replacement

Answer:

- 18
- 17
- 13

10.10 Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware, or explain why it is not possible to do so. If it is possible, calculate what the cost would be.

Answer:
You can use the valid/invalid bit supported in hardware to simulate the reference bit. Initially set the bit to invalid. On first reference, a trap to the operating system is generated. The operating system will set a software bit to 1 and reset the valid/invalid bit to valid.

10.11 You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.

Answer:
No. An optimal algorithm will not suffer from Belady's anomaly because—by definition—an optimal algorithm replaces the page that will not be used for the longest time. Belady's anomaly occurs when a page-replacement algorithm evicts a page that will be needed in the immediate future. An optimal algorithm would not have selected such a page.

10.12 Segmentation is similar to paging but uses variable-sized "pages." Define two segment-replacement algorithms, one based on the FIFO page-replacement scheme and the other on the LRU page-replacement scheme. Remember that since segments are not the same size, the segment that is chosen for replacement may be too small to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated and strategies for systems where they can.

Answer:

a. **FIFO**. Find the first segment large enough to accommodate the incoming segment. If relocation is not possible and no one segment is large enough, select a combination of segments whose memories are contiguous, which are "closest to the first of the list," and which can accommodate the new segment. If relocation is possible, rearrange the memory so that the first *N* segments large enough for the incoming segment are contiguous in memory. Add any leftover space to the free-space list in both cases.

b. **LRU**. Select the segment that has not been used for the longest time and that is large enough, adding any leftover space to the free-space list. If no one segment is large enough, and if relocation is not available, select a combination of the "oldest" segments that are contiguous in memory and are large enough. If relocation is available, rearrange the oldest *N* segments to be contiguous in memory and replace those with the new segment.

10.13 Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of the CPU and the paging disk. Three alternative results are shown below. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

a. CPU utilization 13 percent; disk utilization 97 percent

b. CPU utilization 87 percent; disk utilization 3 percent

c. CPU utilization 13 percent; disk utilization 3 percent

Answer:

a. Thrashing is occurring.

b. CPU utilization is sufficiently high to leave things alone and increase the degree of multiprogramming.

c. Increase the degree of multiprogramming.

10.14 We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can the page table be set up to simulate base and limit registers? How can it be, or why can it not be?

Answer:

The page table can be set up to simulate base and limit registers provided that the memory is allocated in fixed-size segments. The base of a segment can be entered into the page table and the valid/invalid bit used to indicate that portion of the segment as resident in the memory. There will be some problem with internal fragmentation.

Mass-Storage Structure

Practice Exercises

11.1 Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.

Answer:
In a single-user environment, the I/O queue usually is empty. Requests generally arrive from a single process for one block or for a sequence of consecutive blocks. In these cases, FCFS is an economical method of disk scheduling. But LOOK is nearly as easy to program and will give much better performance when multiple processes are performing concurrent I/O, such as when a Web browser retrieves data in the background while the operating system is paging and another application is active in the foreground.

11.2 Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.

Answer:
The center of the disk is the location having the smallest average distance to all other tracks. Thus, the disk head tends to move away from the edges of the disk. Here is another way to think of it. The current location of the head divides the cylinders into two groups. If the head is not in the center of the disk and a new request arrives, the new request is more likely to be in the group that includes the center of the disk; thus, the head is more likely to move in that direction.

11.3 Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?

Answer:
Most disks do not export their rotational position information to the host. Even if they did, the time for this information to reach the scheduler would be subject to imprecision, and the time consumed by the scheduler is variable, so the rotational position information would become incorrect. Further, the disk requests are usually given in terms

of logical block numbers, and the mapping between logical blocks and physical locations is very complex.

11.4 Why is it important to balance file-system I/O among the disks and controllers on a system in a multitasking environment?

Answer:
A system can perform only at the speed of its slowest bottleneck. Disks or disk controllers are frequently the bottlenecks in modern systems, as their individual performance cannot keep up with that of the CPU and system bus. When I/O is balanced among disks and controllers, neither an individual disk nor a controller is overwhelmed, so that bottleneck is avoided.

11.5 What are the tradeoffs involved in rereading code pages from the file system versus using swap space to store them?

Answer:
If code pages are stored in swap space, they can be transferred more quickly to main memory (because swap-space allocation is tuned for faster performance than general file-system allocation). Using swap space can require startup time if the pages are copied there at process invocation rather than just being paged out to swap space on demand. Also, more swap space must be allocated if it is used for both code and data pages.

11.6 Is there any way to implement truly stable storage? Explain your answer.

Answer:
Truly stable storage would never lose data. The fundamental technique for stable storage is to maintain multiple copies of the data, so that if one copy is destroyed, some other copy is still available for use. But for any scheme, we can imagine a large enough disaster that all copies are destroyed.

11.7 It is sometimes said that tape is a sequential-access medium, whereas a hard disk is a random-access medium. In fact, the suitability of a storage device for random access depends on the transfer size. The term ***streaming transfer rate*** denotes the rate for a data transfer that is underway, excluding the effect of access latency. In contrast, the ***effective transfer rate*** is the ratio of total bytes to total seconds, including overhead time such as access latency.

Suppose we have a computer with the following characteristics: the level-2 cache has an access latency of 8 nanoseconds and a streaming transfer rate of 800 megabytes per second, the main memory has an access latency of 60 nanoseconds and a streaming transfer rate of 80 megabytes per second, the hard disk has an access latency of 15 milliseconds and a streaming transfer rate of 5 megabytes per second, and a tape drive has an access latency of 60 seconds and a streaming transfer rate of 2 megabytes per second.

a. Random access causes the effective transfer rate of a device to decrease, because no data are transferred during the access time.

For the disk described, what is the effective transfer rate if an average access is followed by a streaming transfer of (1) 512 bytes, (2) 8 kilobytes, (3) 1 megabyte, and (4) 16 megabytes?

b. The utilization of a device is the ratio of effective transfer rate to streaming transfer rate. Calculate the utilization of the disk drive for each of the four transfer sizes given in part a.

c. Suppose that a utilization of 25 percent (or higher) is considered acceptable. Using the performance figures given, compute the smallest transfer size for a disk that gives acceptable utilization.

d. Complete the following sentence: A disk is a random-access device for transfers larger than _____ bytes and is a sequential-access device for smaller transfers.

e. Compute the minimum transfer sizes that give acceptable utilization for cache, memory, and tape.

f. When is a tape a random-access device, and when is it a sequential-access device?

Answer:

a. For 512 bytes, the effective transfer rate is calculated as follows.
ETR = transfer size/transfer time.
If X is transfer size, then transfer time is ((X/STR) + latency).
Transfer time is 15ms + (512B/5MB per second) = 15.0097ms.
Effective transfer rate is therefore 512B/15.0097ms = 33.12 KB/sec.
ETR for 8KB = .47MB/sec.
ETR for 1MB = 4.65MB/sec.
ETR for 16MB = 4.98MB/sec.

b. Utilization of the device for 512B = 33.12 KB/sec / 5MB/sec = .0064 = .64
For 8KB = 9.4%.
For 1MB = 93%.
For 16MB = 99.6%.

c. Calculate .25 = ETR/STR, solving for transfer size X.
STR = 5MB, so 1.25MB/S = ETR.
1.25MB/S * ((X/5) + .015) = X.
.25X + .01875 = X.
X = .025MB.

d. A disk is a random-access device for transfers larger than K bytes (where K > disk block size), and is a sequential-access device for smaller transfers.

e. Calculate minimum transfer size for acceptable utilization of cache memory:
STR = 800MB, ETR = 200, latency = $8 * 10^{-9}$.
200 (XMB/800 + 8 X 10^{-9}) = XMB.
.25XMB + 1600 * 10^{-9} = XMB.
X = 2.24 bytes.

Calculate for memory:
STR = 80MB, ETR = 20, L = $60 * 10^{-9}$.
20 (XMB/80 + $60 * 10^{-9}$) = XMB.
.25XMB + $1200 * 10^{-9}$ = XMB.
X = 1.68 bytes.
Calculate for tape:
STR = 2MB, ETR = .5, L = 60s.
.5 (XMB/2 + 60) = XMB.
.25XMB + 30 = XMB.
X = 40MB.

f. It depends on how it is being used. Assume we are using the tape to restore a backup. In this instance, the tape acts as a sequential-access device where we are sequentially reading the contents of the tape. As another example, assume we are using the tape to access a variety of records stored on the tape. In this instance, access to the tape is arbitrary and hence considered random.

11.8 Could a RAID level 1 organization achieve better performance for read requests than a RAID level 0 organization (with nonredundant striping of data)? If so, how?

Answer:
Yes, a RAID level 1 organization could achieve better performance for read requests. When a read operation is performed, a RAID level 1 system can decide which of the two copies of the block should be accessed to satisfy the request. It could base this choice on the current location of the disk head and could therefore optimize performance by choosing the disk head that is closer to the target data.

11.9 Give three reasons to use HDDs as secondary storage.

Answer:
HDDs are still the most common secondary storage device.

a. They are the largest random-access storage device for the price, providing terabytes of storage at a low cost.

b. Many devices, from external chassis to storage arrays, are designed to use HDDs, providing a wide variety of ways to use HDDs.

c. They maintain the same read and write performance over their lifetime, unlike NVM storage devices, which lose write performance as they get full and as they age.

11.10 Give three reasons to use NVM devices as secondary storage.

Answer:
NVM devices are increasing in size and decreasing in price faster than HDDs.

a. High-speed NVM devices (including SSDs and usually not including USB drives) are much faster than HDDs. Secondary storage speed has a large impact on overall system performance.

b. NVM devices use less power than HDDs, making them very useful for laptops and other portable, battery-powered devices. NVM devices can also be much smaller than HDDs and thus can, for example, be surface-mounted to motherboards in devices like smartphones.

c. Because NVM devices have no moving parts, they tend to be much more reliable than HDDs.

I/O Systems

Practice Exercises

12.1 State three advantages of placing functionality in a device controller, rather than in the kernel. State three disadvantages.

Answer:
Three advantages:

a. Bugs are less likely to cause an operating-system crash.

b. Performance can be improved by utilizing dedicated hardware and hard-coded algorithms.

c. The kernel is simplified by moving algorithms out of it.

Three disadvantages:

a. Bugs are harder to fix—a new firmware version or new hardware is needed.

b. Improving algorithms requires a hardware update rather than just a kernel or device-driver update.

c. Embedded algorithms could conflict with an application's use of the device, causing decreased performance.

12.2 The example of handshaking in Section 12.2 used two bits: a busy bit and a command-ready bit. Is it possible to implement this handshaking with only one bit? If it is, describe the protocol. If it is not, explain why one bit is insufficient.

Answer:
It is possible, using the following algorithm. Let's assume we simply use the busy bit (or the command-ready bit; this answer is the same regardless). When the bit is off, the controller is idle. The host writes to data-out and sets the bit to signal that an operation is ready (the equivalent of setting the command-ready bit). When the controller is finished, it clears the busy bit. The host then initiates the next operation.

This solution requires that both the host and the controller have read and write access to the same bit, which can complicate circuitry and increase the cost of the controller.

12.3 Why might a system use interrupt-driven I/O to manage a single serial port and polling I/O to manage a front-end processor, such as a terminal concentrator?

Answer:
Polling can be more efficient than interrupt-driven I/O. This is the case when the I/O is frequent and of short duration. Even though a single serial port will perform I/O relatively infrequently and should thus use interrupts, a collection of serial ports such as those in a terminal concentrator can produce a lot of short I/O operations, and interrupting for each one could create a heavy load on the system. A well-timed polling loop could alleviate that load without wasting many resources through looping with no I/O needed.

12.4 Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping, and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than either of the others.

Answer: A hybrid approach could switch between polling and interrupts depending on the length of the I/O operation wait. For example, we could poll and loop N times, and if the device is still busy at N+1, we could set an interrupt and sleep. This approach would avoid long busy-waiting cycles and would be best for very long or very short busy times. It would be inefficient if the I/O completes at N+T (where T is a small number of cycles) due to the overhead of polling plus setting up and catching interrupts.

Pure polling is best with very short wait times. Interrupts are best with known long wait times.

12.5 How does DMA increase system concurrency? How does it complicate hardware design?

Answer:
DMA increases system concurrency by allowing the CPU to perform tasks while the DMA system transfers data via the system and memory buses. Hardware design is complicated because the DMA controller must be integrated into the system, and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and the DMA controller to share use of the memory bus.

12.6 Why is it important to scale up system-bus and device speeds as CPU speed increases?

Answer:

Consider a system that performs 50% I/O and 50% computes. Doubling the CPU performance on this system would increase total system performance by only 50%. Doubling both system aspects would increase performance by 100%. Generally, it is important to remove the current system bottleneck, and to increase overall system performance, rather than blindly increasing the performance of individual system components.

12.7 Distinguish between a driver end and a stream module in a STREAMS operation.

Answer:
The driver end controls a physical device that could be involved in a STREAMS operation. The stream module modifies the flow of data between the stream head (the user interface) and the driver.

File-System Interface

Practice Exercises

13.1 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept. Other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

Answer:
Deleting all files not specifically saved by the user has the advantage of minimizing the file space needed for each user by not saving unwanted or unnecessary files. Saving all files unless specifically deleted is more secure for the user in that the user cannot lose files inadvertently by forgetting to save them.

13.2 Why do some systems keep track of the type of a file, while still others leave it to the user and others simply do not implement multiple file types? Which system is "better"?

Answer:
Some systems allow different file operations based on the type of the file (for instance, an ASCII file can be read as a stream, while a database file can be read via an index to a block). Other systems leave such interpretation of a file's data to the process and provide no help in accessing the data. The method that is "better" depends on the needs of the processes on the system and the demands the users place on the operating system. If a system runs mostly database applications, it may be more efficient for the operating system to implement a database-type file and provide operations, rather than making each program implement the same thing (possibly in different ways). For general-purpose systems, it may be better to implement only basic file types to keep the operating system size smaller and allow maximum freedom to the processes on the system.

13.3 Similarly, some systems support many types of structures for a file's data, while others simply support a stream of bytes. What are the advantages and disadvantages of each approach?

Answer:
An advantage of having the system support different file structures is that the support comes from the system; individual applications are not required to provide the support. In addition, if the system provides the support for different file structures, it can presumably implement the support more efficiently than an application.

The disadvantage of having the system provide support for defined file types is that it increases the size of the system. In addition, applications that require file types other than what is provided by the system may not be able to run on the system.

An alternative strategy is for the operating system to define no support for file structures and instead treat all files as a series of bytes. This is the approach taken by UNIX systems. The advantage of this approach is that it simplifies the operating system support for file systems, as the system no longer has to provide the structure for different file types. Furthermore, it allows applications to define file structures, thereby avoiding the situation in which a system may not provide a file definition required for a specific application.

13.4 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation's success. How would your answer change if file names were limited to seven characters?

Answer:
If arbitrarily long names can be used, then it is possible to simulate a multilevel directory structure. This can be done, for example, by using the character "." to indicate the end of a subdirectory. Thus, for example, the name *jim.java.F1* specifies that *F1* is a file in subdirectory *java*, which in turn is in the root directory *jim*.

If file names were limited to seven characters, then this scheme could not be utilized, and thus, in general, the answer is *no*. The next best approach in this situation would be to use a specific file as a symbol table (directory) to map arbitrarily long names (such as *jim.java.F1*) into shorter arbitrary names (such as *XX00743*), which are then used for actual file access.

13.5 Explain the purpose of the `open()` and `close()` operations.

Answer:

- The `open()` operation informs the system that the named file is about to become active.
- The `close()` operation informs the system that the named file is no longer in active use by the user who issued the close operation.

13.6 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.

a. Describe the protection problems that could arise.

b. Suggest a scheme for dealing with each of these protection problems.

Answer:

a. One piece of information kept in a directory entry is file location. If a user could modify this location, then he could access other files, defeating the access-protection scheme.

b. Do not allow the user to directly write onto the subdirectory. Rather, provide system operations to do so.

13.7 Consider a system that supports 5,000 users. Suppose that you want to allow 4,990 of these users to be able to access one file.

a. How would you specify this protection scheme in UNIX?

b. Can you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?

Answer:

a. There are two methods for achieving this:
   i. Create an access-control list with the names of all 4,990 users.
   ii. Put these 4,990 users in one group, and set the group access accordingly. This scheme cannot always be implemented, since the number of user groups and the number of members per group can be limited by the system.

b. The universal access to files applies to all users unless their names appear in the access-control list with different access permission. Thus, you can simply put the names of the remaining 10 users in the access-control list but give them no access privileges.

13.8 Researchers have suggested that, instead of having an access-control list associated with each file (specifying which users can access the file, and how), we should have a **user control list** associated with each user (specifying which files a user can access, and how). Discuss the relative merits of these two schemes.

Answer:

- *File-based control list*. Since the access-control information is concentrated in one place, it is easier to change the information, and less space is required.
- *User-based control list*. This requires less overhead when opening a file.

CHAPTER 14

File-System Implementation

Practice Exercises

14.1 Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grow at the end. Also assume that the block information to be added is stored in memory.

a. The block is added at the beginning.

b. The block is added in the middle.

c. The block is added at the end.

d. The block is removed from the beginning.

e. The block is removed from the middle.

f. The block is removed from the end.

Answer:
The results are:

	Contiguous	Linked	Indexed
a.	201	1	1
b.	101	52	1
c.	1	3	1
d.	198	1	0
e.	98	52	0
f.	0	100	0

14.2 Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

Answer:

In case of a system crash (memory failure), the free-space list would not be lost, as it would be if the bit map had been stored in main memory.

14.3 Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

Answer:

- **Contiguous**—if file is usually accessed sequentially, if file is relatively small.
- **Linked**—if file is large and usually accessed sequentially.
- **Indexed**—if file is large and usually accessed randomly.

14.4 One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area of a specified size. If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

Answer:
This method requires more overhead then the standard contiguous allocation. It requires less overhead than the standard linked allocation.

14.5 How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

Answer:
Caches allow components of differing speeds to communicate more efficiently by storing data from the slower device, temporarily, in a faster device (the cache). Caches are, almost by definition, more expensive than the devices they are caching for, so increasing the number or size of caches would increase system cost.

14.6 Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

Answer:
Dynamic tables allow more flexibility as a system grows—tables are never exceeded, avoiding artificial use limits. Unfortunately, kernel structures and code are more complicated, so there is more potential for bugs. Dynamic tables use more system resources than static tables, thus potentially taking system resources away from other parts of the system as the system grows.

File-System Internals

Practice Exercises

15.1 Explain how the VFS layer allows an operating system to support multiple types of file systems easily.

Answer:
VFS introduces a layer of indirection in the file system implementation. In many ways, it is similar to object-oriented programming techniques. System calls can be made generically (independent of file system type). Each file system type provides its function calls and data structures to the VFS layer. A system call is translated into the proper specific functions for the target file system at the VFS layer. The calling program has no file-system-specific code, and the upper levels of the system call structures likewise are file system-independent. The translation at the VFS layer turns these generic calls into file-system-specific operations.

15.2 Why have more than one file system type on a given system?

Answer:
File systems can be designed and implemented with specific uses in mind, and optimized for those uses. Consider a virtual memory file system vs. a secondary storage file system. The memory-based one need not concern itself with fragmentation, or persisting data structures in the face of power loss. There are also special-purpose file systems like the procfs file system, designed to give the convenient file system interface to system aspects like the process name space and process resource use.

15.3 On a Unix or Linux system that implements the procfs file system, determine how to use the procfs interface to explore the process name space. What aspects of processes can be viewed via this interface? How would the same information be gathered on a system lacking the procfs file system?

Answer:

On systems containing the procfs psuedo-filesystem, details vary but generally the file system is mounted at /proc, and exploring it with file system commands can reveal the following:

- Each process is represented by its processID, so counting them reveals the number of processes in the system.
- Within each directory under the processID, details of the process state such as its current working directory, command line used to start the process, priority information, memory use information, lock information, open file information, and so on.
- Some procfs also provide interfaces to other kernel structures such as DMA structures, device lists, file system lists and so on.

See the `proc(5)` manual page for details on a given system.

Without procfs, to provide the same information, separate system calls per information type is used, or the ability to open the kernel memory space through `/dev/kmem` or `/dev/sys` is provided. Then programs using these interfaces need to be written to extract the data and present it in human-understandable form. See http://osxbook.com/book/bonus/ancient/procfs for a nice exploration of using procfs vs. not having it available.

15.4 Why do some systems integrate mounted file systems into the root file system naming structure, while others use a separate naming method for mounted file systems?

Answer: As with many aspects of operating system design, choices can be arbitrary or based on some small implementation detail and then exist long after any justify reason. Generally regarding file system mounting, integration with the root file system naming has proven to be more flexible and useful and separate mount point naming and therefore prevails on most file systems.

15.5 Given a remote file access facility such as `ftp`, why were remote file systems like NFS created?

Answer: Users of computer systems value ease-of-use in most cases. The more general purpose, and more widely used and operating system is, the more seamless its operation should be. In the case of remote file access, it is easier for users to use an familiar facility (such as a file system interface) rather than separate commands. And because file systems are tried and true, well integrated, and full featured, existing tools, scripts, and use cases can apply to remote file systems just as local file systems by using a file system interface for remote file access.

CHAPTER 16

Security

Security

Practice Exercises

17.1 What protection problems may arise if a shared stack is used for parameter passing?

Answer:
The contents of the stack could be compromised by any other processes sharing the stack.

17.2 Consider a computing environment where a unique number is associated with each process and each object in the system. Suppose that we allow a process with number n to access an object with number m only if $n > m$. What type of protection structure do we have?

Answer:
A hierarchical structure.

17.3 Consider a computing environment where a process is given the privilege of accessing an object only n times. Suggest a scheme for implementing this policy.

Answer:
Add an integer counter with the capability.

17.4 If all the access rights to an object are deleted, the object can no longer be accessed. At this point, the object should also be deleted, and the space it occupies should be returned to the system. Suggest an efficient implementation of this scheme.

Answer:
Reference counts.

17.5 Why is it difficult to protect a system in which users are allowed to do their own I/O?

Answer:
In earlier chapters, we identified a distinction between kernel and user mode whereby kernel mode is used for carrying out privileged operations such as I/O. One reason why I/O must be performed in kernel

mode is that I/O requires accessing the hardware, and proper access to the hardware is necessary for system integrity. If we allow users to perform their own I/O, we cannot guarantee system integrity.

17.6 Capability lists are usually kept within the address space of the user. How does the system ensure that the user cannot modify the contents of the list?

Answer:
A capability list is considered a "protected object" and is accessed only indirectly by the user. The operating system ensures that the user cannot access the capability list directly.

CHAPTER 18

Virtual Machines

Network Structure

Practice Exercises

19.1 Why would it be a bad idea for routers to pass broadcast packets between networks? What would be the advantages of doing so?

Answer:
All broadcasts would be propagated to all networks, causing a *lot* of network traffic. If broadcast traffic were limited to important data (such as network routing information), then broadcast propagation would save routers from having to run special software to watch for the information and rebroadcast it.

19.2 Discuss the advantages and disadvantages of caching name translations for computers located in remote domains.

Answer:
There is a performance advantage to caching name translations for computers located in remote domains: repeated resolution of the same name from different computers located in the local domain can be performed locally without requiring a remote name-lookup operation. The disadvantage is that there may be inconsistencies in the name translations when updates are made in the mapping of names to IP addresses. These consistency problems can be solved by invalidating translations (by the server telling the caching clients that a change has occurred), which require state to be managed regarding which computers are caching a certain translation and also require a number of invalidation messages. Alternatively, the problems can be solved by using leases, whereby the caching entity invalidates a translation after a certain period of time. The latter approach requires less state and no invalidation messages but might suffer from temporary inconsistencies.

19.3 What are two formidable problems that designers must solve to implement a network system that has the quality of transparency?

Answer:

One issue is making all the processors and storage devices seem transparent across the network. In other words, the distributed system should appear as a centralized system to users. The Andrew file system and NFS provide this feature: the distributed file system appears to the user as a single file system, but in reality it may be distributed across a network.

Another issue concerns the mobility of users. We want to allow users to connect to the "system" rather than to a specific machine (although in reality they may be logging in to a specific machine somewhere in the distributed system).

19.4 To build a robust distributed system, you must know what kinds of failures can occur.

a. List three possible types of failure in a distributed system.

b. Specify which of the entries in your list also are applicable to a centralized system.

Answer:

Three common failures in a distributed system are: (1) network link failure, (2) host failure, (3) site failure failure. Both (2) and (3) are failures that could also occur in a centralized system, whereas a network link failure can occur only in a networked distributed system.

19.5 Is it always crucial to know that the message you have sent has arrived at its destination safely? If your answer is "yes," explain why. If your answer is "no," give appropriate examples.

Answer:

No. Many status-gathering programs work from the assumption that packets may not be received by the destination system. These programs generally *broadcast* a packet and assume that at least some other systems on their network will receive the information. For instance, a daemon on each system might broadcast the system's load average and number of users. This information might be used for process migration target selection. Another example is a program that determines if a remote site is both running and accessible over the network. If the program sends a query and gets no reply, it knows the system cannot currently be reached.

19.6 A distributed system has two sites, A and B. Consider whether site A can distinguish among the following:

a. B goes down.

b. The link between A and B goes down.

c. B is extremely overloaded, and its response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

Answer:

One technique would be for B to periodically send an ***I-am-up*** message to A indicating that it is still alive. If A does not receive an ***I-am-up*** message, it can assume that either B—or the network link—is down. Note that an ***I-am-up*** message does not allow A to distinguish between the types of failure. One technique that allows A to better determine if the network is down is to send an ***Are-you-up*** message to B using an alternate route. If A receives a reply, it can determine that, indeed, the network link is down and B is up.

If we assume that A knows B is up and is reachable (via the ***I-am-up*** mechanism) and that A has some value N that indicates a normal response time, A could monitor the response time from B and compare the values, allowing A to determine if B is overloaded or not.

The implications of both of these techniques are that A could choose another host—say C—in the system if B is either down, unreachable, or overloaded.

CHAPTER 20

The Linux System

Practice Exercises

20.1 Dynamically loadable kernel modules give flexibility when drivers are added to a system, but do they have disadvantages too? Under what circumstances would a kernel be compiled into a single binary file, and when would it be better to keep it split into modules? Explain your answer.

Answer:

There are two principal drawbacks with the use of modules. The first is size: module management consumes unpageable kernel memory, and a basic kernel with a number of modules loaded will consume more memory than an equivalent kernel with the drivers compiled into the kernel image itself. This can be a very significant issue on machines with limited physical memory.

The second drawback is that modules can increase the complexity of the kernel bootstrap process. It is hard to load up a set of modules from disk if the driver needed to access that disk itself a module that needs to be loaded. As a result, managing the kernel bootstrap with modules can require extra work on the part of the administrator: the modules required to bootstrap need to be placed into a ramdisk image that is loaded alongside the initial kernel image when the system is initialized.

In certain cases it is better to use a modular kernel, and in other cases it is better to use a kernel with its device drivers prelinked. Where minimizing the size of the kernel is important, the choice will depend on how often the various device drivers are used. If they are in constant use, then modules are unsuitable. This is especially true where drivers are needed for the boot process itself. On the other hand, if some drivers are not always needed, then the module mechanism allows those drivers to be loaded and unloaded on demand, potentially offering a net saving in physical memory.

Where a kernel is to be built that must be usable on a large variety of very different machines, then building it with modules is clearly preferable to using a single kernel with dozens of unnecessary drivers

consuming memory. This is particularly the case for commercially distributed kernels, where supporting the widest variety of hardware in the simplest manner possible is a priority.

However, if a kernel is being built for a single machine whose configuration is known in advance, then compiling and using modules may simply be an unnecessary complexity. In cases like this, the use of modules may well be a matter of taste.

20.2 Multithreading is a commonly used programming technique. Describe three different ways to implement threads, and compare these three methods with the Linux `clone()` mechanism. When might using each alternative mechanism be better or worse than using clones?

Answer:

Thread implementations can be broadly classified into two groups: kernel-based threads and user-mode threads. User-mode thread packages rely on some kernel support—they may require timer interrupt facilities, for example—but the scheduling between threads is not performed by the kernel but by some library of user-mode code. Multiple threads in such an implementation appear to the operating system as a single execution context. When the multithreaded process is running, it decides for itself which of its threads to execute, using non-local jumps to switch between threads according to its own preemptive or non-preemptive scheduling rules.

Alternatively, the operating system kernel may provide support for threads itself. In this case, the threads may be implemented as separate processes that happen to share a complete or partial common address space, or they may be implemented as separate execution contexts within a single process. Whichever way the threads are organized, they appear as fully independent execution contexts to the application.

Hybrid implementations are also possible, where a large number of threads are made available to the application using a smaller number of kernel threads. Runnable user threads are run by the first available kernel thread.

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.

A lesser advantage is the ability to assign different security attributes to each thread.

User-mode implementations do not have these advantages. Because such implementations run entirely within a single kernel execution context, only one thread can ever be running at once, even if multiple CPUs are available. For the same reason, if one thread enters a system call, no other threads can run until that system call completes. As a result, one thread doing a blocking disk read will hold up every thread in the application. However, user-mode implementations do have their own advantages. The most obvious is performance: invoking the kernel's own scheduler to switch between threads involves entering a new protection domain as the CPU switches to kernel mode, whereas switching between threads in user mode can be achieved simply by saving and restoring the main CPU registers. User-mode threads may also consume less system memory: most UNIX systems will reserve at least a full page for a kernel stack for each kernel thread, and this stack may not be pageable.

The hybrid approach, implementing multiple user threads over a smaller number of kernel threads, allows a balance between these tradeoffs to be achieved. The kernel threads will allow multiple threads to be in blocking kernel calls at once and will permit running on multiple CPUs, and user-mode thread switching can occur within each kernel thread to perform lightweight threading without the overheads of having too many kernel threads. The downside of this approach is complexity: giving control over the tradeoff complicates the thread library's user interface.

20.3 The Linux kernel does not allow paging out of kernel memory. What effect does this restriction have on the kernel's design? What are two advantages and two disadvantages of this design decision?

Answer:

The primary impact of disallowing paging of kernel memory in Linux is that the non-preemptability of the kernel is preserved. Any process taking a page fault, whether in kernel or in user mode, risks being rescheduled while the required data is paged in from disk. Because the kernel can rely on not being rescheduled during access to its primary data structures, locking requirements to protect the integrity of those data structures are very greatly simplified. Although design simplicity is a benefit in itself, it also provides an important performance advantage on uniprocessor machines due to the fact that it is not necessary to do additional locking on most internal data structures.

There are a number of disadvantages to the lack of pageable kernel memory, however. First of all, it imposes constraints on the amount of memory that the kernel can use. It is unreasonable to keep very large data structures in non-pageable memory, since that represents physical memory that absolutely cannot be used for anything else. This has two impacts: first of all, the kernel must prune back many of its internal data structures manually, instead of being able to rely on a single virtual-memory mechanism to keep physical memory usage under control. Second, it makes it infeasible to implement certain features that require large amounts of virtual memory in the kernel, such as the /tmp-

filesystem (a fast virtual-memory-based file system found on some UNIX systems).

Note that the complexity of managing page faults while running kernel code is not an issue here. The Linux kernel code is already able to deal with page faults: it needs to be able to deal with system calls whose arguments reference user memory that may be paged out to disk.

20.4 Discuss three advantages of dynamic (shared) linkage of libraries compared with static linkage. Describe two cases in which static linkage is preferable.

Answer:

The primary advantages of shared libraries are that they reduce the memory and disk space used by a system, and they enhance maintainability.

When shared libraries are being used by all running programs, there is only one instance of each system library routine on disk, and at most one instance in physical memory. When the library in question is one used by many applications and programs, then the disk and memory savings can be quite substantial. In addition, the startup time for running new programs can be reduced, since many of the common functions needed by that program are likely to be already loaded into physical memory.

Maintainability is also a major advantage of dynamic linkage over static. If all running programs use a shared library to access their system library routines, then upgrading those routines, either to add new functionality or to fix bugs, can be done simply by replacing that shared library. There is no need to recompile or relink any applications; any programs loaded after the upgrade is complete will automatically pick up the new versions of the libraries.

There are other advantages too. A program that uses shared libraries can often be adapted for specific purposes simply by replacing one or more of its libraries, or even (if the system allows it, and most UNIXs including Linux do) adding a new one at run time. For example, a debugging library can be substituted for a normal one to trace a problem in an application. Shared libraries also allow program binaries to be linked against commercial, proprietary library code without actually including any of that code in the program's final executable file. This is important because on most UNIX systems, many of the standard shared libraries are proprietary, and licensing issues may prevent including that code in executable files to be distributed to third parties.

In some places, however, static linkage is appropriate. One example is in rescue environments for system administrators. If a system administrator makes a mistake while installing any new libraries, or if hardware develops problems, it is quite possible for the existing shared libraries to become corrupt. As a result, often a basic set of rescue utilities are linked statically, so that there is an opportunity to correct the fault without having to rely on the shared libraries functioning correctly.

There are also performance advantages that sometimes make static linkage preferable in special cases. For a start, dynamic linkage does

increase the startup time for a program, as the linking must now be done at run time rather than at compile time. Dynamic linkage can also sometimes increase the maximum working set size of a program (the total number of physical pages of memory required to run the program). In a shared library, the user has no control over where in the library binary file the various functions reside. Since most functions do not precisely fill a full page or pages of the library, loading a function will usually result in loading in parts of the surrounding functions, too. With static linkage, absolutely no functions that are not referenced (directly or indirectly) by the application need to be loaded into memory.

Other issues surrounding static linkage include ease of distribution: it is easier to distribute an executable file with static linkage than with dynamic linkage if the distributor is not certain whether the recipient will have the correct libraries installed in advance. There may also be commercial restrictions against redistributing some binaries as shared libraries. For example, the license for the UNIX "Motif" graphical environment allows binaries using Motif to be distributed freely as long as they are statically linked, but the shared libraries may not be used without a license.

20.5 Compare the use of networking sockets with the use of shared memory as a mechanism for communicating data between processes on a single computer. What are the advantages of each method? When might each be preferred?

Answer:

Using network sockets rather than shared memory for local communication has a number of advantages. The main advantage is that the socket programming interface features a rich set of synchronization features. A process can easily determine when new data has arrived on a socket connection, how much data is present, and who sent it. Processes can block until new data arrives on a socket, or they can request that a signal be delivered when data arrives. A socket also manages separate connections. A process with a socket open for receive can accept multiple connections to that socket and will be told when new processes try to connect or when old processes drop their connections.

Shared memory offers none of these features. There is no way for a process to determine whether another process has delivered or changed data in shared memory other than by going to look at the contents of that memory. It is impossible for a process to block and request a wakeup when shared memory is delivered, and there is no standard mechanism for other processes to establish a shared memory link to an existing process.

However, shared memory has the advantage that it is very much faster than socket communications in many cases. When data is sent over a socket, it is typically copied from memory to memory multiple times. Shared memory updates require no data copies: if one process updates a data structure in shared memory, that update is immediately visible to all other processes sharing that memory. Sending or receiving data over a socket requires that a kernel system service call be

made to initiate the transfer, but shared memory communication can be performed entirely in user mode with no transfer of control required.

Socket communication is typically preferred when connection management is important or when there is a requirement to synchronize the sender and receiver. For example, server processes will usually establish a listening socket to which clients can connect when they want to use that service. Once the socket is established, individual requests are also sent using the socket, so that the server can easily determine when a new request arrives and who it arrived from.

In some cases, however, shared memory is preferred. Shared memory is often a better solution when either large amounts of data are to be transferred or when two processes need random access to a large common data set. In this case, however, the communicating processes may still need an extra mechanism in addition to shared memory to achieve synchronization between themselves. The X Window System, a graphical display environment for UNIX, is a good example of this: most graphic requests are sent over sockets, but shared memory is offered as an additional transport in special cases where large bitmaps are to be displayed on the screen. In this case, a request to display the bitmap will still be sent over the socket, but the bulk data of the bitmap itself will be sent via shared memory.

20.6 At one time, UNIX systems used disk-layout optimizations based on the rotation position of disk data, but modern implementations, including Linux, simply optimize for sequential data access. Why do they do so? Of what hardware characteristics does sequential access take advantage? Why is rotational optimization no longer so useful?

Answer:

The performance characteristics of disk hardware have changed substantially in recent years. In particular, many enhancements have been introduced to increase the maximum bandwidth that can be achieved on a disk. In a modern system, there can be a long pipeline between the operating system and the disk's read-write head. A disk I/O request has to pass through the computer's local disk controller, over bus logic to the disk drive itself, and then internally to the disk, where there is likely to be a complex controller that can cache data accesses and potentially optimize the order of I/O requests.

Because of this complexity, the time taken for one I/O request to be acknowledged and for the next request to be generated and received by the disk can far exceed the amount of time between one disk sector passing under the read-write head and the next sector header arriving. In order to be able efficiently to read multiple sectors at once, disks will employ a readahead cache. While one sector is being passed back to the host computer, the disk will be busy reading the next sectors in anticipation of a request to read them. If read requests start arriving in an order that breaks this readahead pipeline, performance will drop. As a result, performance benefits substantially if the operating system tries to keep I/O requests in strict sequential order.

A second feature of modern disks is that their geometry can be very complex. The number of sectors per cylinder can vary according to the

position of the cylinder: more data can be squeezed into the longer tracks nearer the edge of the disk than at the center of the disk. For an operating system to optimize the rotational position of data on such disks, it would have to have complete understanding of this geometry, as well as the timing characteristics of the disk and its controller. In general, only the disk's internal logic can determine the optimal scheduling of I/Os, and the disk's geometry is likely to defeat any attempt by the operating system to perform rotational optimizations.