First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes. Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult. The SJF algorithm is a special case of the general priority scheduling algorithm, which simply allocates the CPU to the highest-priority process. Both priority and SJF scheduling may suffer from starvation. Aging is a technique to prevent starvation.

Round-robin (RR) scheduling is more appropriate for a time-shared (interactive) system. RR scheduling allocates the CPU to the first process in the ready queue for \( q \) time units, where \( q \) is the time quantum. After \( q \) time units, if the process has not relinquished the CPU, it is preempted, and the process is put at the tail of the ready queue. The major problem is the selection of the time quantum. If the quantum is too large, RR scheduling degenerates to FCFS scheduling; if the quantum is too small, scheduling overhead in the form of context-switch time becomes excessive.

The FCFS algorithm is nonpreemptive; the RR algorithm is preemptive. The SJF and priority algorithms may be either preemptive or nonpreemptive.

Multilevel queue algorithms allow different algorithms to be used for different classes of processes. The most common model includes a foreground interactive queue that uses RR scheduling and a background batch queue that uses FCFS scheduling. Multilevel feedback queues allow processes to move from one queue to another.

Many contemporary computer systems support multiple processors and allow each processor to schedule itself independently. Typically, each processor maintains its own private queue of processes (or threads), all of which are available to run. Additional issues related to multiprocessor scheduling include processor affinity, load balancing, and multicore processing as well as scheduling on virtualization systems.

Operating systems supporting threads at the kernel level must schedule threads—not processes—for execution. This is the case with Solaris and Windows XP. Both of these systems schedule threads using preemptive, priority-based scheduling algorithms, including support for real-time threads. The Linux process scheduler uses a priority-based algorithm with real-time support as well. The scheduling algorithms for these three operating systems typically favor interactive over batch and CPU-bound processes.

The wide variety of scheduling algorithms demands that we have methods to select among algorithms. Analytic methods use mathematical analysis to determine the performance of an algorithm. Simulation methods determine performance by imitating the scheduling algorithm on a "representative" sample of processes and computing the resulting performance. However, simulation can at best provide an approximation of actual system performance; the only reliable technique for evaluating a scheduling algorithm is to implement the algorithm on an actual system and monitor its performance in a "real-world" environment.

**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 $n$.

5.2 Explain the difference between preemptive and nonpreemptive scheduling.

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.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>$P_2$</td>
<td>0.4</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1.0</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

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

5.5 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 algorithm 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
5.6 Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) 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?

5.7 Distinguish between PCS and SCS scheduling.

5.8 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 the use of LWPs. Furthermore, the system allows program developers to create real-time threads. Is it necessary to bind a real-time thread to an LWP?

Exercises

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

5.10 Discuss how the following pairs of scheduling criteria conflict in certain settings.
   a. CPU utilization and response time
   b. Average turnaround time and maximum waiting time
   c. I/O device utilization and CPU utilization

5.11 Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?
   a. \( \alpha = 0 \) and \( \tau_0 = 100 \) milliseconds
   b. \( \alpha = 0.99 \) and \( \tau_0 = 10 \) milliseconds

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

<table>
<thead>
<tr>
<th>Process ( P_i )</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

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 smaller priority number implies a higher priority), and RR (quantum = 1).
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)?

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

a. First-come, first-served
b. Shortest job first
c. Round robin
d. Priority

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

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

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

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

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

a. The time quantum is 1 millisecond
b. The time quantum is 10 milliseconds

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

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

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

b. What is the algorithm that results from $\alpha < \beta < 0$?
5.18 Explain the differences in how much the following scheduling algorithms discriminate in favor of short processes:

a. FCFS
b. RR
c. Multilevel feedback queues

5.19 Using the Windows XP scheduling algorithm, determine the numeric priority of each of the following threads.

a. A thread in the `REALTIME.PRIORITY_CLASS` with a relative priority of `HIGHEST`
b. A thread in the `NORMAL.PRIORITY_CLASS` with a relative priority of `NORMAL`
c. A thread in the `HIGH.PRIORITY_CLASS` with a relative priority of `ABOVE.NORMAL`

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

a. What is the time quantum (in milliseconds) for a thread with priority 10? With priority 55?
b. Assume that a thread with priority 35 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?
c. Assume that a thread with priority 35 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

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

\[
\text{Priority} = \frac{\text{recent CPU usage}}{2} + \text{base}
\]

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?
The timestamp protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol also ensures freedom from deadlock, because no transaction ever waits.

6.10 Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time. Typically, computer hardware provides several operations that ensure mutual exclusion. However, such hardware-based solutions are too complicated for most developers to use. Semaphores overcome this obstacle. Semaphores can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.

Various synchronization problems (such as the bounded-buffer problem, the readers–writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors. Several language constructs have been proposed to deal with these problems. Monitors provide the synchronization mechanism for sharing abstract data types. A condition variable provides a method by which a monitor procedure can block its execution until it is signaled to continue.

Operating systems also provide support for synchronization. For example, Solaris, Windows XP, and Linux provide mechanisms such as semaphores, mutexes, spinlocks, and condition variables to control access to shared data. The Pthreads API provides support for mutexes and condition variables.

A transaction is a program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed. To ensure atomicity despite system failure, we can use a write-ahead log. All updates are recorded on the log, which is kept in stable storage. If a system crash occurs, the information in the log is used in restoring the state of the updated data items, which is accomplished by use of the undo and redo operations. To reduce the overhead in searching the log after a system failure has occurred, we can use a checkpoint scheme.

To ensure serializability when the execution of several transactions overlaps, we must use a concurrency-control scheme. Various concurrency-control schemes ensure serializability by delaying an operation or aborting the transaction that issued the operation. The most common ones are locking protocols and timestamp ordering schemes.

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.
6.2 **The Cigarette-Smokers Problem.** Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers using Java synchronization.

6.3 Explain why Solaris, Windows XP, and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutexes, semaphores, adaptive mutexes, and condition variables. In each case, explain why the mechanism is needed.

6.4 Describe how volatile, nonvolatile, and stable storage differ in cost.

6.5 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? Describe how the frequency of checkpoints affects:

- System performance when no failure occurs
- The time it takes to recover from a system crash
- The time it takes to recover from a disk crash

6.6 Explain the concept of transaction atomicity.

6.7 Show that some schedules are possible under the two-phase locking protocol but not possible under the timestamp protocol, and vice versa.

**Exercises**

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

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

```java
boolean flag[2]; /* initially false */
int turn;
```
do {
  flag[i] = TRUE;
  while (flag[j]) {
    if (turn == j) {
      flag[i] = false;
      while (turn == j)
        ; // do nothing
      flag[i] = TRUE;
    }
  }

  // critical section
  turn = j;
  flag[i] = FALSE;

  // remainder section
} while (TRUE);

Figure 6.25 The structure of process $P_i$ in Dekker's algorithm.

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

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

enum pstate {idle, want_in, in_cs};

pstate flag[n];

int turn;

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

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

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

6.13 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

6.14 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
do {
  while (TRUE) {
    flag[i] = want_in;
    j = turn;
    
    while (j != i) {
      if (flag[j] != idle) {
        j = turn;
      } else {
        j = (j + 1) % n;
      }
    }
    
    flag[i] = in_cs;
    j = 0;
    
    while ((j < n) && (j == i || flag[j] != in_cs))
      j++;
    
    if ((j >= n) && (turn == i || flag[turn] == idle))
      break;
  }

  // critical section
  j = (turn + 1) % n;
  
  while (flag[j] == idle)
    j = (j + 1) % n;
  
  turn = j;
  flag[i] = idle;

  // remainder section
} while (TRUE);

Figure 6.26  The structure of process P_i in Eisenberg and McGuire's algorithm.

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

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

6.17  Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.
6.18 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

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

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

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

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

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

6.24 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.23 mainly suitable for small portions.
   
   a. Explain why this is true.
   
   b. Design a new scheme that is suitable for larger portions.

6.25 Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.

6.26 How does the `signal()` operation associated with monitors differ from the corresponding operation defined for semaphores?

6.27 Suppose the `signal()` statement can appear only as the last statement in a monitor procedure. Suggest how the implementation described in Section 6.7 can be simplified in this situation.

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

6.29 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than $n$. Write a monitor to coordinate access to the file.
6.30 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?

6.31 Suppose we replace the wait() and signal() operations of monitors with a single construct await(B), where B is a general Boolean expression that causes the process executing it to wait until B becomes true.
   a. Write a monitor using this scheme to implement the readers-writers problem.
   b. Explain why, in general, this construct cannot be implemented efficiently.
   c. What restrictions need to be put on the await statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see Kessels [1977].)

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

6.33 Why do Solaris, Linux, and Windows XP use spinlocks as a synchronization mechanism only on multiprocessor systems and not on single-processor systems?

6.34 In log-based systems that provide support for transactions, updates to data items cannot be performed before the corresponding entries are logged. Why is this restriction necessary?

6.35 Show that the two-phase locking protocol ensures conflict serializability.

6.36 What are the implications of assigning a new timestamp to a transaction that is rolled back? How does the system process transactions that were issued after the rolled-back transaction but that have timestamps smaller than the new timestamp of the rolled-back transaction?

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

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

```c
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```
When a process wishes to obtain a number of resources, it invokes the `decrease_count()` function:

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

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

```c
/* increase available_resources by count */
int increase_count(int count) {
    available_resources += count;
    return 0;
}
```

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

a. Identify the data involved in the race condition.

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

c. Using a semaphore, fix the race condition. It is ok to modify the `decrease_count()` function so that the calling process is blocked until sufficient resources are available.

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

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

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

```c
decrease_count(count);
```
Practice Exercises

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

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

7.3 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 the 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 7.4.4.

7.4 Prove that the safety algorithm presented in Section 7.5.3 requires an order of $m \times n^2$ operations.

7.5 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 10 jobs per deadlock. Each job is worth about $2 (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 in the average execution time per job of about 10 percent. 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?

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

7.7 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 processes that are blocked waiting for resources. If a blocked process has the desired resources, then these resources are taken away from it and are given to the requesting process. The vector of resources for which the blocked process is waiting is increased to include the resources that were taken away.
For example, consider a system with three resource types and the
vector \textit{Available} initialized to (4,2,2). If process \textit{P}_0 asks for (2,2,1), it gets
them. If \textit{P}_1 asks for (1,0,1), it gets them. Then, if \textit{P}_0 asks for (0,0,1), it
is blocked (resource not available). If \textit{P}_2 now asks for (2,0,0), it gets the
available one (1,0,0) and one that was allocated to \textit{P}_0 (since \textit{P}_0 is blocked).
\textit{P}_0's \textit{Allocation} vector goes down to (1,2,1), and its \textit{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.

7.8 Suppose that you have coded the deadlock-avoidance safety algorithm
and now have been asked to implement the deadlock-detection algo-
rithm. Can you do so by simply using the safety algorithm code and
redefining \( \text{Max}_i = \text{Waiting}_i + \text{Allocation}_i \), where \text{Waiting}_i is a vector
specifying the resources for which process \textit{i} is waiting and \text{Allocation}_i
is as defined in Section 7.5? Explain your answer.

7.9 Is it possible to have a deadlock involving only a single process? Explain
your answer.

Exercises

7.10 Consider the traffic deadlock depicted in Figure 7.9.

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

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

7.11 Consider the deadlock situation that can occur in the dining-
philosophers problem when the philosophers obtain the chopsticks one
at a time. Discuss how the four necessary conditions for deadlock hold
in this setting. Discuss how deadlocks could be avoided by eliminating
any one of the four necessary conditions.

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

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

a. Runtime overheads

b. System throughput

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

a. Increase \textit{Available} (new resources added).

b. Decrease \textit{Available} (resource permanently removed from system).

c. Increase \textit{Max} for one process (the process needs or wants more resources than allowed).

d. Decrease \textit{Max} for one process (the process decides it does not need that many resources).

e. Increase the number of processes.

f. Decrease the number of processes.

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

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

a. The maximum need of each process is between one resource and \( m \) resources.

b. The sum of all maximum needs is less than \( m + n \).
Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

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

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

Consider the following snapshot of a system:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D</td>
<td>A B C D</td>
<td>A B C D</td>
<td></td>
</tr>
<tr>
<td>P_0</td>
<td>0 0 1 2</td>
<td>0 0 1 2</td>
<td>1 5 2 0</td>
</tr>
<tr>
<td>P_1</td>
<td>1 0 0 0</td>
<td>1 7 5 0</td>
<td></td>
</tr>
<tr>
<td>P_2</td>
<td>1 3 5 4</td>
<td>2 3 5 6</td>
<td></td>
</tr>
<tr>
<td>P_3</td>
<td>0 6 3 2</td>
<td>0 6 5 2</td>
<td></td>
</tr>
<tr>
<td>P_4</td>
<td>0 0 1 4</td>
<td>0 6 5 6</td>
<td></td>
</tr>
</tbody>
</table>

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 process P_1 arrives for (0,4,2,0), can the request be granted immediately?

What is the optimistic assumption made in the deadlock-detection algorithm? How can this assumption be violated?

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

Modify your solution to Exercise 7.22 so that it is starvation-free.
in memory, however, user memory accesses can be degraded substantially. A TLB can reduce the performance degradation to an acceptable level.

- **Fragmentation.** A multiprogrammed system will generally perform more efficiently if it has a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste, or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.

- **Relocation.** One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory in such a way that the program does not notice the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.

- **Swapping.** Swapping can be added to any algorithm. At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store and later are copied back to main memory. This scheme allows more processes to be run than can be fit into memory at one time.

- **Sharing.** Another means of increasing the multiprogramming level is to share code and data among different users. Sharing generally requires that either paging or segmentation be used to provide small packets of information (pages or segments) that can be shared. Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.

- **Protection.** If paging or segmentation is provided, different sections of a user program can be declared execute-only, read-only, or read-write. This restriction is necessary with shared code or data and is generally useful in any case to provide simple run-time checks for common programming errors.

### Practice Exercises

8.1 Name two differences between logical and physical addresses.

8.2 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.

8.3 Why are page sizes always powers of 2?
8.4 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?

8.5 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 the one page have on the other page?

8.6 Describe a mechanism by which one segment could belong to the address space of two different processes.

8.7 Sharing segments among processes without requiring that they have the same segment number is possible in a dynamically linked segmentation system.
   a. Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.
   b. Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.

8.8 In the IBM/370, memory protection is provided through the use of keys. A key is a 4-bit quantity. Each 2-K block of memory has a key (the storage key) associated with it. The CPU also has a key (the protection key) associated with it. A store operation is allowed only if both keys are equal or if either is zero. Which of the following memory-management schemes could be used successfully with this hardware?
   a. Bare machine
   b. Single-user system
   c. Multiprogramming with a fixed number of processes
   d. Multiprogramming with a variable number of processes
   e. Paging
   f. Segmentation

8.9 Explain the difference between internal and external fragmentation.

8.10 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linkage editor is used to combine multiple object modules into a single program binary. How does the linkage editor change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linkage editor to facilitate the memory-binding tasks of the linkage editor?
8.11 Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

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

8.13 Compare the memory organization schemes of contiguous memory allocation, pure segmentation, and pure paging with respect to the following issues:
   a. External fragmentation
   b. Internal fragmentation
   c. Ability to share code across processes

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

8.15 Compare paging with segmentation with respect to the amount of memory required by the address translation structures in order to convert virtual addresses to physical addresses.

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

8.17 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
   a. 2375
   b. 19366
   c. 30000
   d. 256
   e. 16385
8.18 Consider a logical address space of 32 pages with 1,024 words per page, mapped onto a physical memory of 16 frames.
   a. How many bits are required in the logical address?
   b. How many bits are required in the physical address?

8.19 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

8.20 Consider a paging system with the page table stored in memory.
   a. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?
   b. If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes zero time, if the entry is there.)

8.21 Why are segmentation and paging sometimes combined into one scheme?

8.22 Explain why sharing a reentrant module is easier when segmentation is used than when pure paging is used.

8.23 Consider the following segment table:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>219</td>
<td>600</td>
</tr>
<tr>
<td>1</td>
<td>2300</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1327</td>
<td>580</td>
</tr>
<tr>
<td>4</td>
<td>1952</td>
<td>96</td>
</tr>
</tbody>
</table>

What are the physical addresses for the following logical addresses?
   a. 0,430
   b. 1,10
   c. 2,500
   d. 3,400
   e. 4,112

8.24 What is the purpose of paging the page tables?

8.25 Consider the hierarchical paging scheme used by the VAX architecture. How many memory operations are performed when a user program executes a memory-load operation?
8.26 Compare the segmented paging scheme with the hashed page table scheme for handling large address spaces. Under what circumstances is one scheme preferable to the other?

8.27 Consider the Intel address-translation scheme shown in Figure 8.22.
   a. Describe all the steps taken by the Intel Pentium in translating a logical address into a physical address.
   b. What are the advantages to the operating system of hardware that provides such complicated memory translation?
   c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?

Programming Problems

8.28 Assume that a system has a 32-bit virtual address with a 4-KB page size. Write a C program that is passed a virtual address (in decimal) on the command line and have it output the page number and offset for the given address. As an example, your program would run as follows:

```
./a.out 19986
```

Your program would output:

```
The address 19986 contains:
   page number = 4
   offset = 3602
```

Writing this program will require using the appropriate data type to store 32 bits. We encourage you to use unsigned data types as well.

Wiley Plus

Visit Wiley Plus for
- Source code
- Solutions to practice exercises
- Additional programming problems and exercises
- Labs using an operating system simulator

Bibliographical Notes

Dynamic storage allocation was discussed by Knuth [1973] (Section 2.5), who found through simulation results that first fit is generally superior to best fit. Knuth [1973] also discussed the 50-percent rule.

The concept of paging can be credited to the designers of the Atlas system, which has been described by Kilburn et al. [1961] and by Howarth et al.