TRANSACTIONAL MEMORY (continued)

The advantage of using such a mechanism rather than locks is that the transactional memory system—not the developer—is responsible for guaranteeing atomicity. Additionally, the system can identify which statements in atomic blocks can be executed concurrently, such as concurrent read access to a shared variable. It is, of course, possible for a programmer to identify these situations and use reader-writer locks, but the task becomes increasingly difficult as the number of threads within an application grows.

Transactional memory can be implemented in either software or hardware. Software transactional memory (STM), as the name suggests, implements transactional memory exclusively in software—no special hardware is needed. STM works by inserting instrumentation code inside transaction blocks. The code is inserted by a compiler and manages each transaction by examining where statements may run concurrently and where specific low-level locking is required. Hardware transactional memory (HTM) uses hardware-cache hierarchies and cache-coherency protocols to manage and resolve conflicts involving shared data residing in separate processors caches. HTM requires no special code instrumentation and thus has less overhead than STM. However, HTM does require that existing cache hierarchies and cache coherency protocols be modified to support transactional memory.

Transactional memory has existed for several years without widespread implementation. However, the growth of multicore systems and the associated emphasis on concurrent programming have prompted a significant amount of research in this area on the part of both academics and hardware vendors, including Intel and Sun Microsystems.

6.9 Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. We discussed this issue briefly in Section 6.5.3 in connection with semaphores, although we will see that deadlocks can occur with many other types of resources available in a computer system.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: “When two
6.9 Deadlocks

trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

6.9.1 System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. Release. The process releases the resource.

The request and release of resources are system calls, as explained in Chapter 2. Examples are the request() and release() device, open() and close() file, and allocate() and free() memory system calls. Request and release of resources that are not managed by the operating system can be accomplished through the wait() and signal() operations on semaphores or through acquisition and release of a mutex lock. For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource. A system table records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The
events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors). However, other types of events may result in deadlocks (for example, the IPC facilities discussed in Chapter 3).

To illustrate a deadlocked state, consider a system with three CD RW drives. Suppose each of three processes holds one of these drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event “CD RW is released,” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process $P_i$ is holding the DVD and process $P_j$ is holding the printer. If $P_i$ requests the printer and $P_j$ requests the DVD drive, a deadlock occurs.

A programmer who is developing multithreaded applications must pay particular attention to this problem. Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

### 6.9.2 Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

**DEADLOCK WITH MUTEX LOCKS**

Let’s see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```c
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two` run in the functions `do_work_one()` and `do_work_two()`, respectively, as shown in Figure 6.22.

(continued on following page.)
DEADLOCK WITH MUTEX LOCKS (continued)

```c
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
    * Do some work
    */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
    * Do some work
    */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Figure 6.22  Deadlock example.

In this example, thread_one attempts to acquire the mutex locks in the order (1) first_mutex, (2) second_mutex, while thread_two attempts to acquire the mutex locks in the order (1) second_mutex, (2) first_mutex. Deadlock is possible if thread_one acquires first_mutex while thread_two acquires second_mutex.

Note that, even though deadlock is possible, it will not occur if thread_one is able to acquire and release the mutex locks for first_mutex and second_mutex before thread_two attempts to acquire the locks. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain circumstances.

### 6.9.2.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion.** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another
2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait.** A set \( \{P_0, P_1, ..., P_n\} \) of waiting processes must exist such that \( P_0 \) is waiting for a resource held by \( P_1 \), \( P_1 \) is waiting for a resource held by \( P_2 \), ..., \( P_{n-1} \) is waiting for a resource held by \( P_n \), and \( P_n \) is waiting for a resource held by \( P_0 \).

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

### 6.9.2.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices \( V \) and a set of edges \( E \). The set of vertices \( V \) is partitioned into two different types of nodes: \( P = \{P_1, P_2, ..., P_n\} \), the set consisting of all the active processes in the system, and \( R = \{R_1, R_2, ..., R_m\} \), the set consisting of all resource types in the system.

A directed edge from process \( P_i \) to resource type \( R_j \) is denoted by \( P_i \rightarrow R_j \); it signifies that process \( P_i \) has requested an instance of resource type \( R_j \) and is currently waiting for that resource. A directed edge from resource type \( R_j \) to process \( P_i \) is denoted by \( R_j \rightarrow P_i \); it signifies that an instance of resource type \( R_j \) has been allocated to process \( P_i \). A directed edge \( P_i \rightarrow R_j \) is called a request edge; a directed edge \( R_j \rightarrow P_i \) is called an assignment edge.

Pictorially, we represent each process \( P_i \) as a circle and each resource type \( R_j \) as a rectangle. Since resource type \( R_j \) may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle \( R_j \), whereas an assignment edge must also designate one of the dots in the rectangle.

When process \( P_i \) requests an instance of resource type \( R_j \), a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure 6.23 depicts the following situation:

- The sets \( P, R, \) and \( E \):
  - \( P = \{P_1, P_2, P_3\} \)
  - \( R = \{R_1, R_2, R_3, R_4\} \)
  - \( E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_2, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_3 \rightarrow P_1, R_3 \rightarrow P_3\} \)
6.9 Deadlocks

Figure 6.23 Resource-allocation graph.

- Resource instances:
  - One instance of resource type $R_1$
  - Two instances of resource type $R_2$
  - One instance of resource type $R_3$
  - Three instances of resource type $R_4$

- Process states:
  - Process $P_1$ is holding an instance of resource type $R_2$ and is waiting for an instance of resource type $R_1$.
  - Process $P_2$ is holding an instance of $R_1$ and an instance of $R_2$ and is waiting for an instance of $R_3$.
  - Process $P_3$ is holding an instance of $R_3$.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 6.23. Suppose that process $P_3$ requests an instance of resource type $R_2$. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph (Figure 6.24). At this point, two minimal cycles exist in the system:
Processes $P_1$, $P_2$, and $P_3$ are deadlocked. Process $P_2$ is waiting for the resource $R_3$, which is held by process $P_3$. Process $P_3$ is waiting for either process $P_1$ or process $P_2$ to release resource $R_2$. In addition, process $P_1$ is waiting for process $P_2$ to release resource $R_1$.

Now consider the resource-allocation graph in Figure 6.25. In this example, we also have a cycle:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

However, there is no deadlock. Observe that process $P_4$ may release its instance of resource type $R_2$. That resource can then be allocated to $P_3$, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or
6.9 Deadlocks

may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

6.9.3 Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

Next, we elaborate briefly on each of the three methods for handling deadlocks. Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods for ensuring that at least one of the necessary conditions (Section 6.9.2.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

**Deadlock avoidance** requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred).

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system’s performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned
earlier. In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods, which must be used constantly. Also, in some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority (or any process running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

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 deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. There are three principal methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- Allow the system to enter a deadlocked state, detect it, and then recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows.

A deadlock can occur only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no preemption, and circular
wait. To prevent deadlocks, we can ensure that at least one of the necessary conditions never holds.

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

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

Exercises

6.6 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.

6.7 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P₀ and P₁, 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 6.26 The structure of process $P_i$ in Dekker’s algorithm.

The structure of process $P_i$ ($i = 0$ or $1$) is shown in Figure 6.26; the other process is $P_j$ ($j = 1$ or $0$). Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.8 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:

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

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 6.27. Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.9 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.

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

6.11 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.

6.12 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 6.27  The structure of process $P_i$ in Eisenberg and McGuire's algorithm.

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

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

6.15 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.
6.16 Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

6.17 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.

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

6.19 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.

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

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

6.22 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.21 mainly suitable for small portions.
   a. Explain why this is true.
   b. Design a new scheme that is suitable for larger portions.


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

6.25 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.

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

6.27 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.

**6.28** 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?

**6.29** Suppose we replace the \texttt{wait()} and \texttt{signal()} operations of monitors with a single construct \texttt{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 \texttt{await} statement so that it can be implemented efficiently? (Hint: Restrict the generality of \( B \); see Kessels [1977].)

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

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

**6.32** 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 \textit{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:

\[
\#define \texttt{MAX\_RESOURCES} 5 \\
\texttt{int available\_resources = MAX\_RESOURCES;}
\]

When a process wishes to obtain a number of resources, it invokes the \texttt{decrease\_count()} function:
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
    if (available_resources < count)
        return -1;
    else {
        available_resources -= count;
        return 0;
    }
}

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

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

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

a. Identify the data involved in the race condition.
b. Identify the location (or locations) in the code where the race condition occurs.
c. Using a semaphore, fix the race condition. It is OK to modify the decrease_count() function so that the calling process is blocked until sufficient resources are available.

6.33 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);
```

The process will return from this function call only when sufficient resources are available.
6.34 Consider the traffic deadlock depicted in Figure 6.28.
   a. Show that the four necessary conditions for deadlock hold in this example.
   b. State a simple rule for avoiding deadlocks in this system.

6.35 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.

Programming Problems

6.36 The Sleeping-Barber Problem. A barbershop consists of a waiting room with \( n \) chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

Programming Projects

Producer–Consumer Problem

In Section 6.6.1, we presented a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, we will design a
#include "buffer.h"

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
    /* insert item into buffer
    return 0 if successful, otherwise
    return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
    /* remove an object from buffer
    placing it in item
    return 0 if successful, otherwise
    return -1 indicating an error condition */
}

Figure 6.29 A skeleton program.

programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 6.10 and 6.11. The solution presented in Section 6.6.1 uses three semaphores: empty and full, which count the number of empty and full slots in the buffer, and mutex, which is a binary (or mutual-exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, standard counting semaphores will be used for empty and full, and a mutex lock, rather than a binary semaphore, will be used to represent mutex. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with these empty, full, and mutex structures. You can solve this problem using either Pthreads or the Win32 API.

The Buffer

Internally, the buffer will consist of a fixed-size array of type buffer_item (which will be defined using a typedef). The array of buffer_item objects will be manipulated as a circular queue. The definition of buffer_item, along with the size of the buffer, can be stored in a header file such as the following:

    /* buffer.h */
    typedef int buffer_item;
    #define BUFFER_SIZE 5

The buffer will be manipulated with two functions, insert_item() and remove_item(), which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 6.29.

The insert_item() and remove_item() functions will synchronize the producer and consumer using the algorithms outlined in Figures 6.10 and


#include "buffer.h"

int main(int argc, char *argv[]) {
    /* 2. Initialize buffer */
    /* 3. Create producer thread(s) */
    /* 4. Create consumer thread(s) */
    /* 5. Sleep */
    /* 6. Exit */
}

Figure 6.30  A skeleton program.

6.11. The buffer will also require an initialization function that initializes the mutual-exclusion object mutex along with the empty and full semaphores.

The main() function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the main() function will sleep for a period of time and, upon awakening, will terminate the application. The main() function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears in Figure 6.30.

Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the rand() function, which produces random integers between 0 and RAND_MAX. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears in Figure 6.31.

In the following sections, we first cover details specific to Pthreads and then describe details of the Win32 API.

Pthreads Thread Creation

Creating threads using the Pthreads API is discussed in Section 4.3.1. Please refer to that Section for specific instructions regarding creation of the producer and consumer using Pthreads.

Pthreads Mutex Locks

The code sample depicted in Figure 6.32 illustrates how mutex locks available in the Pthread API can be used to protect a critical section.
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

void *producer(void *param) {
    buffer_item item;
    while (TRUE) {
        /* sleep for a random period of time */
        sleep(...);
        /* generate a random number */
        item = rand();
        if (insert_item(item))
            fprintf("report error condition");
        else
            printf("producer produced %d\n",item);
    }
}

void *consumer(void *param) {
    buffer_item item;
    while (TRUE) {
        /* sleep for a random period of time */
        sleep(...);
        if (remove_item(&item))
            fprintf("report error condition");
        else
            printf("consumer consumed %d\n",item);
    }
}

Figure 6.31 An outline of the producer and consumer threads.

Pthreads uses the pthread_mutex_t data type for mutex locks. A mutex is created with the pthread_mutex_init(&mutex,NULL) function, with the first parameter being a pointer to the mutex. By passing NULL as a second parameter, we initialize the mutex to its default attributes. The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions. If the mutex lock is unavailable when pthread_mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_unlock(). All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

Pthreads Semaphores

Pthreads provides two types of semaphores—named and unnamed. For this project, we use unnamed semaphores. The code below illustrates how a semaphore is created:
#include <pthread.h>
pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex,NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/*** critical section ***/

/* release the mutex lock */
pthread_mutex_unlock(&mutex);

Figure 6.32 Code sample.

#include <semaphore.h>

sem_t sem;

/* Create the semaphore and initialize it to 5 */
sem_init(&sem, 0, 5);

The sem_init() creates and initializes a semaphore. This function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore’s initial value

In this example, by passing the flag 0, we are indicating that this semaphore can be shared only by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to the value 5.

In Section 6.5, we described the classical wait() and signal() semaphore operations. Pthreads names the wait() and signal() operations sem_wait() and sem_post(), respectively. The code sample shown in Figure 6.33 creates a binary semaphore mutex with an initial value of 1 and illustrates its use in protecting a critical section.

Win32

Details concerning thread creation using the Win32 API are available in Section 4.3.2. Please refer to that Section for specific instructions.
```
#include <semaphore.h>
sem_t mutex;

/* create the semaphore */
sem_init(&mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&mutex);

/*** critical section ***/

/* release the semaphore */
sem_post(&mutex);
```

Figure 6.33  Code example.

Win32 Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 6.8.2. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

The first parameter refers to a security attribute for the mutex lock. By setting this attribute to NULL, we are disallowing any children of the process creating this mutex lock to inherit the handle of the mutex. The second parameter indicates whether the creator of the mutex is the initial owner of the mutex lock. Passing a value of FALSE indicates that the thread creating the mutex is not the initial owner; we shall soon see how mutex locks are acquired. The third parameter allows naming of the mutex. However, because we provide a value of NULL, we do not name the mutex. If successful, `CreateMutex()` returns a HANDLE to the mutex lock; otherwise, it returns NULL.

In Section 6.8.2, we identified dispatcher objects as being either signaled or nonsignaled. A signaled object is available for ownership; once a dispatcher object (such as a mutex lock) is acquired, it moves to the nonsignaled state. When the object is released, it returns to signaled.

Mutex locks are acquired by invoking the `WaitForSingleObject()` function, passing the function the HANDLE to the lock and a flag indicating how long to wait. The following code demonstrates how the mutex lock created above can be acquired:

```
WaitForSingleObject(Mutex, INFINITE);
```

The parameter value INFINITE indicates that we will wait an infinite amount of time for the lock to become available. Other values could be used that would allow the calling thread to time out if the lock did not become available within
a specified time. If the lock is in a signaled state, `WaitForSingleObject()` returns immediately, and the lock becomes nonsignaled. A lock is released (moves to the signaled state) by invoking `ReleaseMutex()`, such as:

```c
ReleaseMutex(Mutex);
```

**Win32 Semaphores**

Semaphores in the Win32 API are also dispatcher objects and thus use the same signaling mechanism as mutex locks. Semaphores are created as follows:

```c
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

The first and last parameters identify a security attribute and a name for the semaphore, similar to what was described for mutex locks. The second and third parameters indicate the initial value and maximum value of the semaphore. In this instance, the initial value of the semaphore is 1 and its maximum value is 5. If successful, `CreateSemaphore()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

Semaphores are acquired with the same `WaitForSingleObject()` function as mutex locks. We acquire the semaphore `Sem` created in this example by using the statement:

```c
WaitForSingleObject(Semaphore, INFINITE);
```

If the value of the semaphore is > 0, the semaphore is in the signaled state and thus is acquired by the calling thread. Otherwise, the calling thread blocks indefinitely—as we are specifying `INFINITE`—until the semaphore becomes signaled.

The equivalent of the `signal()` operation on Win32 semaphores is the `ReleaseSemaphore()` function. This function is passed three parameters:

1. The `HANDLE` of the semaphore
2. The amount by which to increase the value of the semaphore
3. A pointer to the previous value of the semaphore

We can increase `Sem` by 1 using the following statement:

```c
ReleaseSemaphore(Sem, 1, NULL);
```

Both `ReleaseSemaphore()` and `ReleaseMutex()` return nonzero if successful and zero otherwise.
Bibliographical Notes

The mutual-exclusion problem was first discussed in a classic paper by Dijkstra [1965a]. Dekker’s algorithm (Exercise 6.7)—the first correct software solution to the two-process mutual-exclusion problem—was developed by the Dutch mathematician T. Dekker. This algorithm also was discussed by Dijkstra [1965a]. A simpler solution to the two-process mutual-exclusion problem has since been presented by Peterson [1981] (Figure 6.2).

Dijkstra [1965b] presented the first solution to the mutual-exclusion problem for \( n \) processes. This solution, however, does not have an upper bound on the amount of time a process must wait before it is allowed to enter the critical section. Knuth [1966] presented the first algorithm with a bound; his bound was \( 2^n \) turns. A refinement of Knuth’s algorithm by deBruijn [1967] reduced the waiting time to \( n^2 \) turns, after which Eisenberg and McGuire [1972] succeeded in reducing the time to the lower bound of \( n-1 \) turns. Another algorithm that also requires \( n-1 \) turns but is easier to program and to understand is the bakery algorithm, which was developed by Lamport [1974]. Burns [1978] developed the hardware-solution algorithm that satisfies the bounded-waiting requirement.

General discussions concerning the mutual-exclusion problem were offered by Lamport [1986] and Lamport [1991]. A collection of algorithms for mutual exclusion was given by Raynal [1986].

The semaphore concept was suggested by Dijkstra [1965a]. Patil [1971] examined the question of whether semaphores can solve all possible synchronization problems. Parnas [1975] discussed some of the flaws in Patil’s arguments. Kosaraju [1973] followed up on Patil’s work to produce a problem that cannot be solved by \texttt{wait()} and \texttt{signal()} operations. Lipton [1974] discussed the limitations of various synchronization primitives.

The classic process-coordination problems that we have described are paradigms for a large class of concurrency-control problems. The bounded-buffer problem, the dining-philosophers problem, and the sleeping-barber problem (Exercise 6.36) were suggested by Dijkstra [1965a] and Dijkstra [1971]. The cigarette-smokers problem (Exercise 6.2) was developed by Patil [1971]. The readers–writers problem (Exercise 6.2) was developed by Patil [1971]. The issue of concurrent reading and writing was discussed by Lamport [1977]. The problem of synchronization of independent processes was discussed by Lamport [1976].

The critical-region concept was suggested by Hoare [1972] and by Brinch-Hansen [1972]. The monitor concept was developed by Brinch-Hansen [1973]. A complete description of the monitor was given by Hoare [1974]. Kessels [1977] proposed an extension to the monitor to allow automatic signaling. Experience obtained from the use of monitors in concurrent programs was discussed by Lamport and Redell [1979]. They also examined the priority-inversion problem. General discussions concerning concurrent programming were offered by Ben-Ari [1990] and Birrell [1989].

Optimizing the performance of locking primitives has been discussed in many works, such as Lamport [1987], Mellor-Crummey and Scott [1991], and Anderson [1990]. The use of shared objects that do not require the use of critical sections was discussed in Herlihy [1993], Bershad [1993], and Kopetz and Reisinger [1993]. Novel hardware instructions and their utility in implementing
synchronization primitives have been described in works such as Culler et al. [1998], Goodman et al. [1989], Barnes [1993], and Herlihy and Moss [1993]. Some details of the locking mechanisms used in Solaris were presented in Mauro and McDougall [2007]. Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside the kernel. Details of Windows 2000 synchronization can be found in Solomon and Russinovich [2000]. Goetz et al. [2006] presents a detailed discussion of concurrent programming in Java as well as the java.util.concurrent package.

Dijkstra [1965a] was one of the first and most influential contributors in the deadlock area. A more recent study of deadlock handling is provided in Levine [2003]. Adl-Tabatabai et al. [2007] discuss transactional memory.