CHAPTER 6

Process Synchronization

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

CHAPTER OBJECTIVES

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- To present both software and hardware solutions of the critical-section problem.
- To explore classical problems in process synchronization.

6.1 Background

In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer–consumer problem, which is representative of operating systems. Specifically, in Section 3.4.1, we described how a bounded buffer can be used to enable processes to share memory.

Let's return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at most BUFFER_SIZE − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable `counter`, initialized to 0. `counter` is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (true) {
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

Although both the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable `counter` is currently 5 and that the producer and consumer processes execute the statements "`counter++`" and "`counter--`" concurrently. Following the execution of these two statements, the value of the variable `counter` may be 4, 5, or 6! The only correct result, though, is `counter` == 5, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `counter` may be incorrect as follows. Note that the statement "`counter++`" may be implemented in machine language (on a typical machine) as

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

where $register_1$ is one of the local CPU registers. Similarly, the statement $register_2$"`counter--`" is implemented as follows:

$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

where again $register_2$ is one of the local CPU registers. Even though $register_1$ and $register_2$ may be the same physical register (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

The concurrent execution of "`counter++`" and "`counter--`" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Notice that we have arrived at the incorrect state "`counter` == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5, we would arrive at the incorrect state "`counter` == 6".

We would arrive at this incorrect state because we allowed both processes to manipulate the variable `counter` concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can manipulate the variable `counter`. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, with the growth of multicore systems, there is an increased emphasis on developing multithreaded applications wherein several threads—which are quite possibly sharing data—are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, a major portion of this chapter is concerned with **process synchronization** and **coordination** among cooperating processes.

6.2 The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P_i is shown in

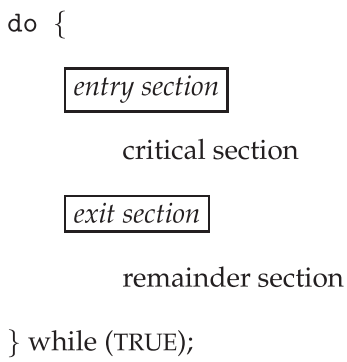

Figure 6.1 General structure of a typical process P_i.

Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the **relative speed** of the n processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (*kernel code*) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode

to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. Furthermore, a preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. Of course, this effect can be minimized by designing kernel code that does not behave in this way. Later in this chapter, we explore how various operating systems manage preemption within the kernel.

6.3 Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as `load` and `store`, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, `j` equals $1 - $ `i`.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable `turn` indicates whose turn it is to enter its critical section. That is, if `turn == i`, then process P_i is allowed to execute in its critical section. The `flag` array is used to indicate if a process *is ready* to enter its critical section. For example, if `flag[i]` is `true`, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 6.2.

To enter the critical section, process P_i first sets `flag[i]` to `true` and then sets `turn` to the value `j`, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, `turn` will be set to both `i` and `j` at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.

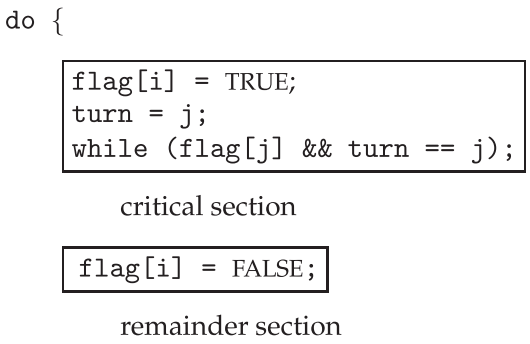

Figure 6.2 The structure of process P_i in Peterson's solution.

The eventual value of `turn` determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either `flag[j] == false` or `turn == i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0] == flag[1] == true`. These two observations imply that P_0 and P_1 could not have successfully executed their `while` statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes —say, P_j —must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("`turn == j`"). However, at that time, `flag[j] == true` and `turn == j`, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition `flag[j] == true` and `turn == j`; this loop is the only one possible. If P_j is not ready to enter the critical section, then `flag[j] == false`, and P_i can enter its critical section. If P_j has set `flag[j]` to `true` and is also executing in its `while` statement, then either `turn == i` or `turn == j`. If `turn == i`, then P_i will enter the critical section. If `turn == j`, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since P_i does not change the value of the variable `turn` while executing the `while` statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Figure 6.3 Solution to the critical-section problem using locks.

6.4 Synchronization Hardware

We have just described one software-based solution to the critical-section problem. However, as mentioned, software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Instead, we can generally state that any solution to the critical-section problem requires a simple tool—a **lock**. Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated in Figure 6.3.

In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to application programmers. All these solutions are based on the premise of locking; however, as we shall see, the designs of such locks can be quite sophisticated.

We start by presenting some simple hardware instructions that are available on many systems and showing how they can be used effectively in solving the critical-section problem. Hardware features can make any programming task easier and improve system efficiency.

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 6.4 The definition of the `TestAndSet()` instruction.

```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

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

Figure 6.5 Mutual-exclusion implementation with `TestAndSet()`.

message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock if the clock is kept updated by interrupts.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the `TestAndSet()` and `Swap()` instructions.

The `TestAndSet()` instruction can be defined as shown in Figure 6.4. The important characteristic of this instruction is that it is executed atomically. Thus, if two `TestAndSet()` instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the `TestAndSet()` instruction, then we can implement mutual exclusion by declaring a Boolean variable `lock`, initialized to `false`. The structure of process P_i is shown in Figure 6.5.

The `Swap()` instruction, in contrast to the `TestAndSet()` instruction, operates on the contents of two words; it is defined as shown in Figure 6.6. Like the `TestAndSet()` instruction, it is executed atomically. If the machine supports the `Swap()` instruction, then mutual exclusion can be provided as follows. A global Boolean variable `lock` is declared and is initialized to `false`. In addition, each process has a local Boolean variable `key`. The structure of process P_i is shown in Figure 6.7.

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. In Figure 6.8, we present another algorithm using the `TestAndSet()` instruction that satisfies all the critical-section requirements. The common data structures are

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

Figure 6.6 The definition of the `Swap()` instruction.

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

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

Figure 6.7 Mutual-exclusion implementation with the `Swap()` instruction.

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to `false`. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section only if either `waiting[i] == false` or `key == false`. The value of `key` can become `false` only if the `TestAndSet()` is executed. The first process to execute the `TestAndSet()` will find `key == false`; all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, maintaining the mutual-exclusion requirement.

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

      // critical section

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

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

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

Figure 6.8 Bounded-waiting mutual exclusion with `TestAndSet()`.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array `waiting` in the cyclic ordering ($i + 1, i + 2, ..., n - 1, 0, ..., i - 1$). It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Unfortunately for hardware designers, implementing atomic `TestAndSet()` instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

6.5 Semaphores

The hardware-based solutions to the critical-section problem presented in Section 6.4 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a **semaphore**.

A semaphore `S` is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. The `wait()` operation was originally termed `P` (from the Dutch *proberen*, "to test"); `signal()` was originally called `V` (from *verhogen*, "to increment"). The definition of `wait()` is as follows:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

The definition of `signal()` is as follows:

```
signal(S) {
    S++;
}
```

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of `S` ($S \leq 0$), as well as its possible modification (`S--`), must be executed without interruption. We shall see how these operations can be implemented in Section 6.5.2; first, let us see how semaphores can be used.

6.5.1 Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some

systems, binary semaphores are known as **mutex locks**, as they are locks that provide *mut*ual *ex*clusion.

We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, `mutex`, initialized to 1. Each process P_i is organized as shown in Figure 6.9.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore `synch`, initialized to 0, and by inserting the statements

```
S1;
signal(synch);
```

in process P_1 and the statements

```
wait(synch);
S2;
```

in process P_2. Because `synch` is initialized to 0, P_2 will execute S_2 only after P_1 has invoked `signal(synch)`, which is after statement S_1 has been executed.

6.5.2 Implementation

The main disadvantage of the semaphore definition given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system,

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

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

Figure 6.9 Mutual-exclusion implementation with semaphores.

where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)

To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations. When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore `S`, should be restarted when some other process executes a `signal()` operation. The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct {
     int value;
     struct process *list;
} semaphore;
```

Each semaphore has an integer `value` and a list of processes `list`. When a process must wait on a semaphore, it is added to the list of processes. A `signal()` operation removes one process from the list of waiting processes and awakens that process.

The `wait()` semaphore operation can now be defined as

```
wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
                 add this process to S->list;
                 block();
          }
}
```

The `signal()` semaphore operation can now be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

The `block()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a blocked process `P`. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, although semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the `wait()` operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use *any* queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists.

It is vital that semaphores be executed atomically. We must guarantee that no two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment (that is, where only one CPU exists), we can solve it by simply inhibiting interrupts during the time the `wait()` and `signal()` operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multiprocessor environment, interrupts must be disabled on every processor; otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques—such as spinlocks—to ensure that `wait()` and `signal()` are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait()` and `signal()` operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `wait()` and `signal()` operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or

even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

6.5.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, we consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, `S` and `Q`, set to the value 1:

```
        P0                  P1
    wait(S);            wait(Q);
    wait(Q);            wait(S);
       .                   .
       .                   .
       .                   .
    signal(S);          signal(Q);
    signal(Q);          signal(S);
```

Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlock 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.* However, other types of events may result in deadlocks, which we discuss in Section 6.9.

Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

6.5.4 Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority. As an example, assume we have three processes, L, M, and H, whose priorities follow the order $L < M < H$. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process

PRIORITY INVERSION AND THE MARS PATHFINDER

Priority inversion can be more than a scheduling inconvenience. On systems with tight time constraints (such as real-time systems), priority inversion can cause a process to take longer than it should to accomplish a task. When that happens, other failures can cascade, resulting in system failure.

Consider the Mars Pathfinder, a 1997 NASA space probe that landed a robot, the Sojourner rover, on Mars to conduct experiments. Shortly after the Sojourner began operating, it started to experience frequent computer resets. Each reset reinitialized all hardware and software, including communications. If the problem had not been solved, the Sojourner would have failed in its mission.

The problem was caused by the fact that one high-priority task, "bc_dist," was taking longer than expected to complete its work. This task was being forced to wait for a shared resource that was held by the lower-priority "ASI/MET" task, which in turn was preempted by multiple medium-priority tasks. The "bc_dist" task would stall waiting for the shared resource, and ultimately the "bc_sched" task would discover the problem and perform a reset. The Sojourner was suffering from a typical case of priority inversion.

The operating system on the Sojourner was VxWorks, which had a global variable to enable priority inheritance on all semaphores. After testing, the variable was set on the Sojourner (on Mars!), and the problem was solved.

A full description of the priority-inversion problem, its detection, and its solution was written by the software team lead and is available at research.microsoft.com/mbj/Mars_Pathfinder/Authoritative_Account.html.

L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to relinquish resource R.

This problem is known as **priority inversion**. It occurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however. Typically these systems solve the problem by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority. Because resource R would now be available, process H—not M—would run next.

6.6 Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for

```
do {
      . . .
   // produce an item in nextp
      . . .
   wait(empty);
   wait(mutex);
      . . .
   // add nextp to buffer
      . . .
   signal(mutex);
   signal(full);
} while (TRUE);
```

Figure 6.10 The structure of the producer process.

testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization.

6.6.1 The Bounded-Buffer Problem

The *bounded-buffer problem* was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation; we provide a related programming project in the exercises at the end of the chapter.

We assume that the pool consists of n buffers, each capable of holding one item. The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers. The semaphore `empty` is initialized to the value `n`; the semaphore `full` is initialized to the value 0.

The code for the producer process is shown in Figure 6.10; the code for the consumer process is shown in Figure 6.11. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
} while (TRUE);
```

Figure 6.11 The structure of the consumer process.

6.6.2 The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the *readers–writers problem*. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers–writers problem. Refer to the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

The semaphores `mutex` and `wrt` are initialized to 1; `readcount` is initialized to 0. The semaphore `wrt` is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `readcount` is updated. The `readcount` variable keeps track of how many processes are currently reading the object. The semaphore `wrt` functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 6.12; the code for a reader process is shown in Figure 6.13. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `wrt`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(wrt)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Figure 6.12 The structure of a writer process.

requires specifying the mode of the lock: either *read* or *write* access. When a process wishes only to read shared data, it requests the reader–writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

6.6.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

Figure 6.13 The structure of a reader process.

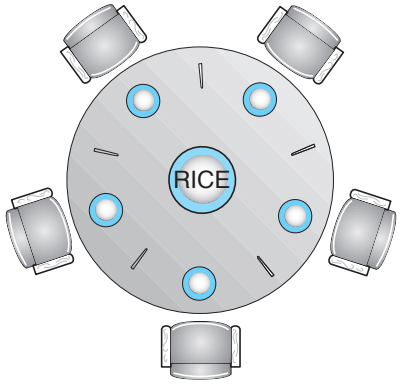

Figure 6.14 The situation of the dining philosophers.

to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 6.14). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The *dining-philosophers problem* is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore; she releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 6.15.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next.

- Allow at most four philosophers to be sitting simultaneously at the table.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
} while (TRUE);
```

Figure 6.15 The structure of philosopher *i*.

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

6.7 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

We have seen an example of such errors in the use of counters in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the counter value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable `mutex`, which is initialized to 1. Each process must execute `wait(mutex)` before entering the critical section and `signal(mutex)` afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we examine the various difficulties that may result. Note that these difficulties will arise even if a *single* process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a process interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution:

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

  In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

  In this case, a deadlock will occur.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models discussed in Section 6.6.

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

6.7.1 Usage

A *abstract data type* — or ADT — encapsulates private data with public methods to operate on that data. A *monitor type* is an ADT that presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables. The syntax of a monitor type is shown in Figure 6.16. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Figure 6.16 Syntax of a monitor.

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.17). However, the monitor construct, as defined so far, is not sufficiently powerful to model some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the `condition` construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal()` operation has no effect; that is, the state of `x` is the same as if the operation had never been executed (Figure

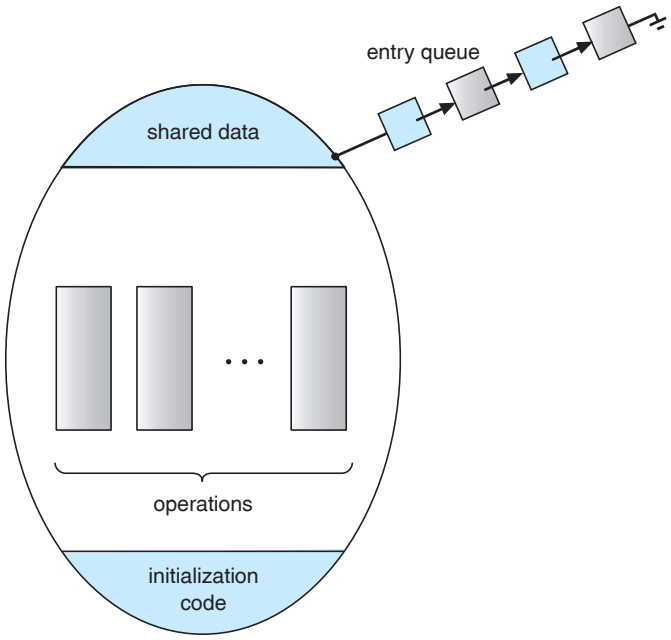

Figure 6.17 Schematic view of a monitor.

6.18). Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore.

Now suppose that, when the `x.signal()` operation is invoked by a process `P`, there exists a suspended process `Q` associated with condition `x`. Clearly, if the suspended process `Q` is allowed to resume its execution, the signaling process `P` must wait. Otherwise, both `P` and `Q` would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. **Signal and wait**. *P* either waits until *Q* leaves the monitor or waits for another condition.
2. **Signal and continue**. *Q* either waits until *P* leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since *P* was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other hand, if we allow thread *P* to continue, then by the time *Q* is resumed, the logical condition for which *Q* was waiting may no longer hold. A compromise between these two choices was adopted in the language Concurrent Pascal. When thread *P* executes the signal operation, it immediately leaves the monitor. Hence, *Q* is immediately resumed.

Many programming languages have incorporated the idea of the monitor as described in this section, including Concurrent Pascal, Mesa, C# (pronounced *C-sharp*), and Java. Other languages—such as Erlang—provide some type of concurrency support using a similar mechanism.

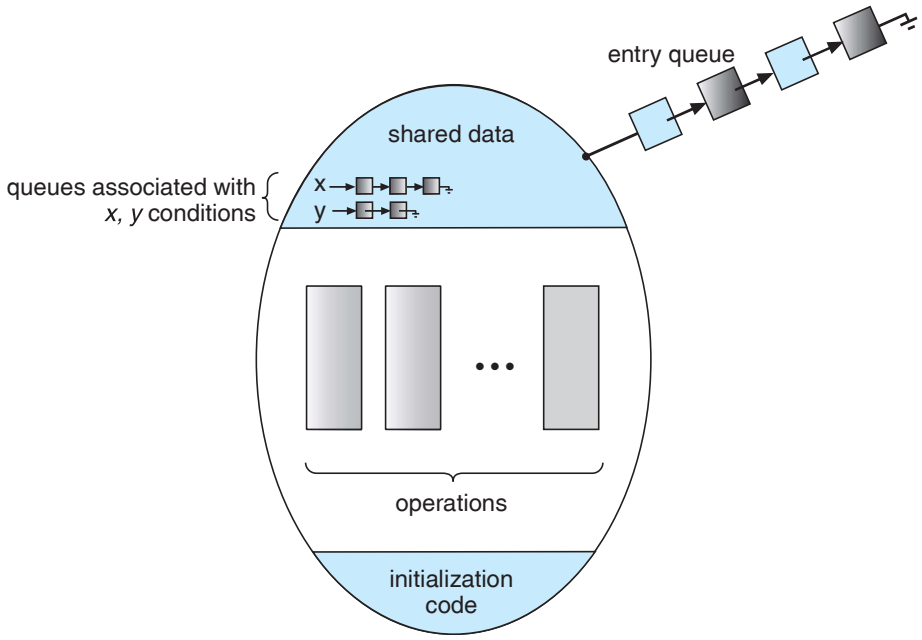

Figure 6.18 Monitor with condition variables.

6.7.2 Dining-Philosophers Solution Using Monitors

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher *i* can set the variable `state[i] = EATING` only if her two neighbors are not eating: (`state[(i+4) % 5] != EATING`) and (`state[(i+1) % 5] != EATING`).

We also need to declare

```
condition self[5];
```

in which philosopher *i* can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor `DiningPhilosophers`, whose definition is shown in Figure 6.19. Each philosopher, before starting to eat, must invoke the operation `pickup()`. This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Figure 6.19 A monitor solution to the dining-philosopher problem.

the `putdown()` operation. Thus, philosopher *i* must invoke the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
                ...
                eat
                ...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. We do not present a solution to this problem but rather leave it as an exercise for you.

6.7.3 Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore `mutex` (initialized to 1) is provided. A process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, `next`, is introduced, initialized to 0. The signaling processes can use `next` to suspend themselves. An integer variable `next_count` is also provided to count the number of processes suspended on `next`. Thus, each external procedure `F` is replaced by

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented as well. For each condition `x`, we introduce a semaphore `x_sem` and an integer variable `x_count`, both initialized to 0. The operation `x.wait()` can now be implemented as

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

The operation `x.signal()` can be implemented as

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen. In some cases, however, the generality of the implementation is unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.25.

6.7.4 Resuming Processes Within a Monitor

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition `x`, and an `x.signal()` operation

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

Figure 6.20 A monitor to allocate a single resource.

is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use an FCFS ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used; it has the form

```
x.wait(c);
```

where `c` is an integer expression that is evaluated when the `wait()` operation is executed. The value of `c`, which is called a **priority number**, is then stored with the name of the process that is suspended. When `x.signal()` is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the `ResourceAllocator` monitor shown in Figure 6.20, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
    ...
  access the resource;
    ...
R.release();
```

where `R` is an instance of type `ResourceAllocator`.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the `ResourceAllocator` monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the `ResourceAllocator` monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only through the use of the additional mechanisms that are described in Chapter 13.

6.8 Synchronization Examples

We next describe the synchronization mechanisms provided by the Solaris, Windows, and Linux operating systems, as well as the Pthreads API. We have chosen these three operating systems because they provide good examples of different approaches to synchronizing the kernel, and we have included the Pthreads API because it is widely used for thread creation and synchronization by developers on UNIX and Linux systems. As you will see in this section, the synchronization methods available in these differing systems vary in subtle and significant ways.

JAVA MONITORS

Java provides a monitor-like concurrency mechanism for thread synchronization. Every object in Java has associated with it a single lock. When a method is declared to be `synchronized`, calling the method requires owning the lock for the object. We declare a `synchronized` method by placing the `synchronized` keyword in the method definition. The following defines the `safeMethod()` as `synchronized`, for example:

```
public class SimpleClass {
   . . .
   public synchronized void safeMethod() {
    /* Implementation of safeMethod() */
    . . .
   }
}
```

Next, assume we create an object instance of `SimpleClass`, such as:

```
SimpleClass sc = new SimpleClass();
```

Invoking the `sc.safeMethod()` method requires owning the lock on the object instance `sc`. If the lock is already owned by another thread, the thread calling the `synchronized` method blocks and is placed in the **entry set** for the object's lock. The entry set represents the set of threads waiting for the lock to become available. If the lock is available when a `synchronized` method is called, the calling thread becomes the owner of the object's lock and can enter the method. The lock is released when the thread exits the method; a thread from the entry set is then selected as the new owner of the lock.

Java also provides `wait()` and `notify()` methods, which are similar in function to the `wait()` and `signal()` statements for a monitor. Release 1.5 of the Java language provides API support for semaphores, condition variables, and mutex locks (among other concurrency mechanisms) in the `java.util.concurrent` package.

6.8.1 Synchronization in Solaris

To control access to critical sections, Solaris provides adaptive mutexes, condition variables, semaphores, reader–writer locks, and turnstiles. Solaris implements semaphores and condition variables essentially as they are presented in Sections 6.5 and 6.7. In this section, we describe adaptive mutexes, reader–writer locks, and turnstiles.

An **adaptive mutex** protects access to every critical data item. On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked and therefore already in use, the adaptive mutex does one of two things. If the lock is held by a thread that is currently running on another CPU, the thread spins while waiting for the lock to become available, because the thread holding the lock is likely to finish

soon. If the thread holding the lock is not currently in run state, the thread blocks, going to sleep until it is awakened by the release of the lock. It is put to sleep so that it will not spin while waiting, since the lock will not be freed very soon. A lock held by a sleeping thread is likely to be in this category. On a single-processor system, the thread holding the lock is never running if the lock is being tested by another thread, because only one thread can run at a time. Therefore, on this type of system, threads always sleep rather than spin if they encounter a lock.

Solaris uses the adaptive-mutex method to protect only data that are accessed by short code segments. That is, a mutex is used if a lock will be held for less than a few hundred instructions. If the code segment is longer than that, the spin-waiting method is exceedingly inefficient. For these longer code segments, condition variables and semaphores are used. If the desired lock is already held, the thread issues a wait and sleeps. When a thread frees the lock, it issues a signal to the next sleeping thread in the queue. The extra cost of putting a thread to sleep and waking it, and of the associated context switches, is less than the cost of wasting several hundred instructions waiting in a spinlock.

Reader–writer locks are used to protect data that are accessed frequently but usually in a read-only manner. In these circumstances, reader–writer locks are more efficient than semaphores, because multiple threads can read data concurrently, whereas semaphores always serialize access to the data. Reader–writer locks are relatively expensive to implement, so again they are used only on long sections of code.

Solaris uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or a reader–writer lock. A **turnstile** is a queue structure containing threads blocked on a lock. For example, if one thread currently owns the lock for a synchronized object, all other threads trying to acquire the lock will block and enter the turnstile for that lock. When the lock is released, the kernel selects a thread from the turnstile as the next owner of the lock. Each synchronized object with at least one thread blocked on the object's lock requires a separate turnstile. However, rather than associating a turnstile with each synchronized object, Solaris gives each kernel thread its own turnstile. Because a thread can be blocked only on one object at a time, this is more efficient than having a turnstile for each object.

The turnstile for the first thread to block on a synchronized object becomes the turnstile for the object itself. Threads subsequently blocking on the lock will be added to this turnstile. When the initial thread ultimately releases the lock, it gains a new turnstile from a list of free turnstiles maintained by the kernel. To prevent a priority inversion, turnstiles are organized according to a **priority-inheritance protocol**. This means that if a lower-priority thread currently holds a lock on which a higher-priority thread is blocked, the thread with the lower priority will temporarily inherit the priority of the higher-priority thread. Upon releasing the lock, the thread will revert to its original priority.

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. A crucial implementation difference is the priority-inheritance protocol. Kernel-locking routines adhere to the kernel priority-inheritance methods used by the scheduler; user-level thread-locking mechanisms do not provide this functionality.

To optimize Solaris performance, developers have refined and fine-tuned the locking methods. Because locks are used frequently and typically are used for crucial kernel functions, tuning their implementation and use can produce great performance gains.

6.8.2 Synchronization in Windows XP

The Windows operating system is a multithreaded kernel that provides support for real-time applications and multiple processors. When the Windows kernel accesses a global resource on a uniprocessor system, it temporarily masks interrupts for all interrupt handlers that may also access the global resource. On a multiprocessor system, Windows protects access to global resources using spinlocks. Just as in Solaris, the kernel uses spinlocks only to protect short code segments. Furthermore, for reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows provides **dispatcher objects**. Using a dispatcher object, threads synchronize according to several different mechanisms, including mutexes, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. Semaphores behave as described in Section 6.5. **Events** are similar to condition variables; that is, they are used to notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired.

Dispatcher objects may be in either a signaled state or a nonsignaled state. A **signaled state** indicates that an object is available and a thread will not block when acquiring the object. A **nonsignaled state** indicates that an object is not available and a thread will block when attempting to acquire the object. We illustrate the state transitions of a mutex lock dispatcher object in Figure 6.21.

A relationship exists between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object. If so, the kernel moves one thread—or possibly more—from the waiting state to the ready state, where they can resume executing. The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object for which it is waiting. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be "owned" by only a single

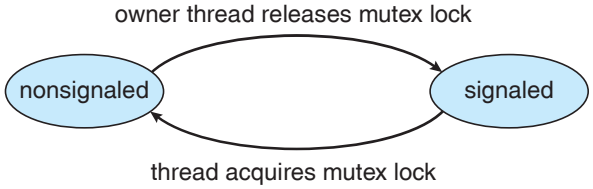

Figure 6.21 Mutex dispatcher object.

thread. For an event object, the kernel will select all threads that are waiting for the event.

We can use a mutex lock as an illustration of dispatcher objects and thread states. If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object. When the mutex moves to the signaled state (because another thread has released the lock on the mutex), the thread waiting at the front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

We provide a programming project at the end of this chapter that uses mutex locks and semaphores in the Win32 API.

6.8.3 Synchronization in Linux

Prior to Version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on single-processor machines, rather than holding a spinlock, the kernel disables kernel preemption; and rather than releasing the spinlock, it enables kernel preemption. This is summarized below:

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple system calls—`preempt_disable()` and `preempt_enable()`—for disabling and enabling kernel preemption. In addition, however, the kernel is not preemptible if a kernel-mode task is holding a lock. To enforce this rule, each task in the system has a `thread-info` structure containing a counter, `preempt_count`, to indicate the number of locks being held by the task. When a lock is acquired, `preempt_count` is incremented. It is decremented when a lock is released. If the value of `preempt_count` for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted (assuming there are no outstanding calls to `preempt_disable()`).

Spinlocks—along with enabling and disabling kernel preemption—are used in the kernel only when a lock (or disabling kernel preemption) is held for a short duration. When a lock must be held for a longer period, semaphores are appropriate.

6.8.4 Synchronization in Pthreads

The Pthreads API provides mutex locks, condition variables, and read–write locks for thread synchronization. This API is available for programmers and is not part of any particular kernel. Mutex locks are the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Condition variables in Pthreads behave much as described in Section 6.7. Read–write

TRANSACTIONAL MEMORY

With the emergence of multicore systems has come increased pressure to develop multithreaded applications that take advantage of multiple processing cores. However, multithreaded applications present an increased risk of race conditions and deadlocks. Traditionally, techniques such as locks, semaphores, and monitors have been used to address these issues. However, **transactional memory** provides an alternative strategy for developing thread-safe concurrent applications.

A **memory transaction** is a sequence of memory read–write operations that are atomic. If all operations in a transaction are completed, the memory transaction is committed; otherwise, the operations must be aborted and rolled back. The benefits of transactional memory can be obtained through features added to a programming language.

Consider an example. Suppose we have a function `update()` that modifies shared data. Traditionally, this function would be written using locks such as the following:

```
update () {
   acquire();
   /* modify shared data */
   release();
}
```

However, using synchronization mechanisms such as locks and semaphores involves many potential problems, including deadlocks. Additionally, as the number of threads increases, traditional locking does not scale well.

As an alternative to traditional methods, new features that take advantage of transactional memory can be added to a programming language. In our example, suppose we add the construct `atomic{S}`, which ensures that the operations in `S` execute as a transaction. This allows us to rewrite the `update()` method as follows:

```
update () {
   atomic {
      /* modify shared data */
   }
}
```

(*continued on following page*)

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.

locks behave similarly to the locking mechanism described in Section 6.6.2. Many systems that implement Pthreads also provide semaphores, although they are not part of the Pthreads standard and instead belong to the POSIX SEM extension. Other extensions to the Pthreads API include spinlocks, but not all extensions are considered portable from one implementation to another. We provide a programming project at the end of this chapter that uses Pthreads mutex locks and semaphores.

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

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:

```
/* 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)

```
/* 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` aacquites `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

process requests that resource, the requesting process must be delayed until the resource has been released.

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:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

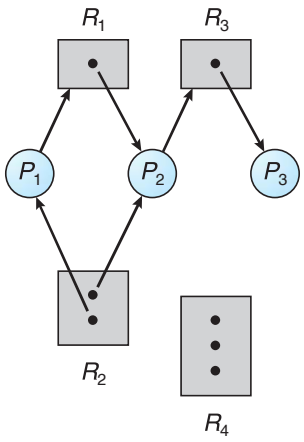

Figure 6.23 Resource-allocation graph.

- 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:

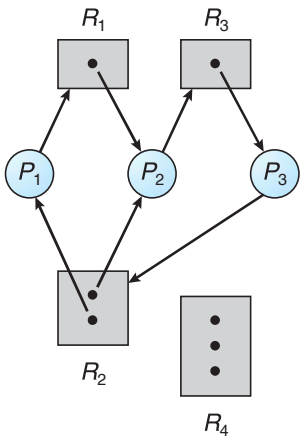

Figure 6.24 Resource-allocation graph with a deadlock.

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

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

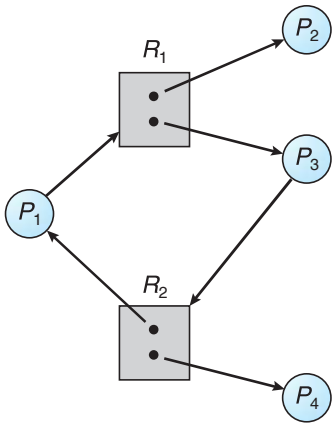

Figure 6.25 Resource-allocation graph with a cycle but no deadlock.

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