Process Synchronization

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

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 to the critical-section problem.
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity.

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 could be used to enable processes to share memory.

Let's return to our consideration of the bounded buffer. Here, we assume that the code for the producer is as follows:

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
++count;
```

The code for the consumer is

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
--count;
```

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

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

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

where $register_1$ is a local CPU register. Similarly, the statement "`--count`" is implemented as follows:

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

where again $register_2$ is a local CPU register. 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 "`++count`" and "`--count`" 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	$register_1 =$ `count`	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 =$ `count`	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	`count` $= register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	`count` $= register_2$	$\{count = 4\}$

Notice that we have arrived at the incorrect state "`count` == 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 "`count` == 6".

We would arrive at this incorrect state because we allowed both processes to manipulate the variable `count` 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 be manipulating the variable `count`. 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. 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.

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Figure 6.1 General structure of a typical process P_i.

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 be `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. 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.

```
while (true) {

      flag[i] = TRUE;
      turn = j;
      while (flag[j] && turn == j);

            critical section

      flag[i] = FALSE;

            remainder section

}
```

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

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 persists 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).

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. 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.

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

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

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 single-processor 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 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. The `HardwareData` class shown in Figure 6.4 illustrates the instructions.

The `getAndSet()` method implementing the *get-and-set* instruction is shown in Figure 6.4. The important characteristic of this instruction is that it is executed atomically. Thus, if two *get-and-set* instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

```
public class HardwareData
{
   private boolean value = false;

   public HardwareData(boolean value) {
      this.value = value;
   }

   public boolean get() {
      return value;
   }

   public void set(boolean newValue) {
      value = newValue;
   }

   public boolean getAndSet(boolean newValue) {
      boolean oldValue = this.get();
      this.set(newValue);

      return oldValue;
   }

  public void swap(HardwareData other) {
     boolean temp = this.get();

     this.set(other.get());
     other.set(temp);
   }
}
```

Figure 6.4 Data structure for hardware solutions.

If the machine supports the *get-and-set* instruction, then we can implement mutual exclusion by declaring `lock` to be an object of class `HardwareData` and initializing it to `false`. All threads will share access to the `lock` object. Figure 6.5 illustrates the structure of an arbitrary thread. Notice that this thread uses the `yield()` method introduced in Section 5.7. Invoking `yield()` keeps the thread in the runnable state but also allows the JVM to select another runnable thread to run.

The *swap* instruction, defined in the `swap()` method in Figure 6.4, operates on the contents of two words; like the *get-and-set* instruction, it is executed atomically. If the machine supports the *swap* instruction, then mutual exclusion can be provided as follows. All threads share an object `lock` of class `HardwareData` that is initialized to `false`. In addition, each thread has a local `HardwareData` object `key`. The structure of an arbitrary thread is shown in Figure 6.6.

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

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
   while (lock.getAndSet(true))
      Thread.yield();

   // critical section
   lock.set(false);
   // remainder section
}
```

Figure 6.5 Thread using *get-and-set* lock.

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* contains an integer variable that, apart from initialization, is accessed only through two standard operations: `acquire()` and `release()`. These operations were originally termed `P` (from the Dutch *proberen*, meaning "to test") and `V` (from *verhogen*, meaning "to increment").

Assuming `value` represents the integer value of the semaphore, the definitions of `acquire()` and `release()` are shown in Figure 6.7. Modifications to the integer value of the semaphore in the `acquire()` and `release()` operations must be executed indivisibly. That is, when one thread modifies the semaphore value, no other thread can simultaneously modify that same semaphore value.

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
   key.set(true);

   do {
      lock.swap(key);
   }
   while (key.get() == true);

   // critical section
   lock.set(false);
   // remainder section
}
```

Figure 6.6 Thread using *swap* instruction.

```
acquire() {
   while (value <= 0)
      ; // no-op
   value--;
}

release() {
   value++;
}
```

Figure 6.7 The definitions of `acquire()` and `release()`.

In addition, in the case of `acquire()`, the testing of the integer value of the semaphore (`value` <= 0) and of its possible modification (`value--`) must be executed without interruption. We will see in Section 6.5.2 how these operations can be implemented; first, let's 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 mutual exclusion.

We can use binary semaphores to control access to the critical section for a process or thread. The general strategy is as follows (assuming that the semaphore is initialized to 1):

```
Semaphore sem = new Semaphore(1);

sem.acquire();

   // critical section

sem.release();

   // remainder section
```

A generalized solution for multiple threads is shown in the Java program in Figure 6.8. Five separate threads are created, but only one can be in its critical section at a given time. The semaphore `sem`, which is shared by all the threads, controls access to the critical section.

Counting semaphores can be used for a variety of purposes, such as controlling access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each thread that wishes to use a resource performs an `acquire()` operation on the semaphore (thereby decrementing the count). When a thread releases a resource, it performs a `release()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, threads that wish to use a resource will block until the count becomes greater than 0.

```
public class Worker implements Runnable
{
   private Semaphore sem;

   public Worker(Semaphore sem) {
      this.sem = sem;
   }

   public void run() {
      while (true) {
         sem.acquire();
         criticalSection();
         sem.release();
         remainderSection();
      }
   }
}

public class SemaphoreFactory
{
   public static void main(String args[]) {
      Semaphore sem = new Semaphore(1);
      Thread[] bees = new Thread[5];

      for (int i = 0; i < 5; i++)
         bees[i] = new Thread(new Worker(sem));
      for (int i = 0; i < 5; i++)
         bees[i].start();
   }
}
```

Figure 6.8 Synchronization using semaphores.

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;
synch.release();
```

in process P_1 and the statements

```
synch.acquire();
S2;
```

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

6.5.2 Implementation

The main disadvantage of the semaphore definition just described 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 multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. A semaphore that produces this result is also called a **spinlock**, because the process "spins" while waiting for the lock. (Spinlocks have the advantage 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 definitions of the `acquire()` and `release()` semaphore operations. When a process executes the `acquire()` operation and finds that the semaphore value is not positive, it must wait. However, rather than using 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 `release()` 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 (1) an integer value and (2) a list of processes. When a process must wait on a semaphore, it is added to the list of processes for that semaphore. The `release()` operation removes one process from the list of waiting processes and awakens that process.

The semaphore operations can now be defined as

```
acquire(){
    value--;
    if (value < 0) {
       add this process to list
       block();
    }
}

release(){
    value++;
    if (value <= 0) {
       remove a process P from 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 the semaphore value is never negative under the classical definition of semaphores with busy waiting. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact is a result of switching the order of the decrement and the test in the implementation of the `acquire()` 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, which ensures 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 may use *any* queueing strategy. Correct use of semaphores does not depend on a particular queueing strategy for the semaphore lists.

The critical aspect of semaphores is that they be executed atomically. We must guarantee that no two processes can execute `acquire()` and `release()` operations on the same semaphore at the same time. This situation creates a critical-section problem, which can be solved in one of two ways.

In a single-processor environment, we can simply inhibit interrupts during the time the `acquire()` and `release()` operations are executing. 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, however, inhibiting interrupts does not work. Instructions from different processes (running on different processors) may be interleaved in some arbitrary way. If the hardware does not provide any special instructions, we can employ any of the correct software solutions for the critical-section problem (Section 6.2), where the critical sections consist of the `acquire()` and `release()` operations.

It is important to admit we have not completely eliminated busy waiting with this definition of the `acquire()` and `release()` operations. Rather, we have moved busy waiting to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `acquire()` and `release()` operations. These sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied; 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 this case, busy waiting is extremely inefficient. Throughout this chapter, we address issues of performance and show techniques to avoid busy waiting. In Section 6.8.7.2, we show how semaphores are provided in the Java API.

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 `release()` operation. When such a state is reached, these processes are said to be **deadlocked**.

As an illustration, 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

S.acquire();      Q.acquire();
Q.acquire();      S.acquire();
      .                 .
      .                 .
      .                 .
S.release();      Q.release();
Q.release();      S.release();
```

Suppose that P_0 executes `S.acquire()`, and then P_1 executes `Q.acquire()`. When P_0 executes `Q.acquire()`, it must wait until P_1 executes `Q.release()`. Similarly, when P_1 executes `S.acquire()`, it must wait until P_0 executes `S.release()`. 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, as we show in Chapter 7. In that chapter, we describe various mechanisms for dealing with the deadlock problem.

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 add and remove processes from the list associated with a semaphore in last-in, first-out (LIFO) 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 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

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—see Chapter 19), 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 NASA space probe that landed a robot, the Sojourner rover, on Mars in 1997 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 the reset. The Sojourner was suffering from a typical case of priority inversion.

The operating system on the Sojourner was VxWorks (see Section 19.6), 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 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.

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 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. A solution is shown in Figure 6.9. A producer places an item in the buffer by calling the `insert()` method (Figure 6.10), and consumers remove items by invoking `remove()` (Figure 6.11).

The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to 1. The `empty` and `full` semaphores count the number of empty and full buffers. The semaphore `empty` is initialized to the capacity of the buffer—`BUFFER_SIZE`. The semaphore `full` is initialized to 0.

The producer thread is shown in Figure 6.12. The producer alternates between sleeping for a while (the `SleepUtilities` class is available in WileyPLUS), producing a message, and attempting to place that message into the buffer via the `insert()` method.

The consumer thread is shown in Figure 6.13. The consumer alternates between sleeping and consuming an item using the `remove()` method.

```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;
    private E[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
      // buffer is initially empty
      in = 0;
      out = 0;
      mutex = new Semaphore(1);
      empty = new Semaphore(BUFFER_SIZE);
      full = new Semaphore(0);

      buffer = (E[]) new Object[BUFFER_SIZE];

    }

    public void insert(E item) {
      // Figure 6.10
    }

    public E remove() {
      // Figure 6.11
    }
}
```

Figure 6.9 Solution to the bounded-buffer problem using semaphores.

```
// Producers call this method
public void insert(E item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```

Figure 6.10 The `insert()` method.

The `Factory` class (Figure 6.14) creates the producer and consumer threads, passing each a reference to the `BoundedBuffer` object.

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. This requirement leads to the **readers–writers problem**. Since it was originally stated, this problem has been

```
// Consumers call this method
public E remove() {
    E item;

    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

Figure 6.11 The `remove()` method.

```
import java.util.Date;

public class Producer implements Runnable
{
   private Buffer<Date> buffer;

   public Producer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
      }
   }
}
```

Figure 6.12 The producer.

```
import java.util.Date;

public class Consumer implements Runnable
{
   private Buffer<Date> buffer;

   public Consumer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // consume an item from the buffer
         message = (Date)buffer.remove();
      }
   }
}
```

Figure 6.13 The consumer.

```
import java.util.Date;

public class Factory
{
   public static void main(String args[]) {
      Buffer<Date> buffer = new BoundedBuffer<Date>();

      // Create the producer and consumer threads
      Thread producer = new Thread(new Producer(buffer));
      Thread consumer = new Thread(new Consumer(buffer));

      producer.start();
      consumer.start();
   }
}
```

Figure 6.14 The `Factory` class.

used to test nearly every new synchronization primitive. The 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 database. 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

```
public class Reader implements Runnable
{
   private ReadWriteLock db;

   public Reader(ReadWriteLock db) {
      this.db = db;
   }

   public void run() {
      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         db.acquireReadLock();

         // now read from the database
         SleepUtilities.nap();

         db.releaseReadLock();
      }
   }
}
```

Figure 6.15 A reader.

```
public class Writer implements Runnable
{
   private ReadWriteLock db;

   public Writer(ReadWriteLock db) {
      this.db = db;
   }

   public void run() {
      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         db.acquireWriteLock();

         // now write to write to the database
         SleepUtilities.nap();

         db.releaseWriteLock();
      }
   }
}
```

Figure 6.16 A writer.

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 can 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. Here, we present the Java class files for a solution to the first readers–writers problem. It does not address starvation. (In the exercises at the end of the chapter, you are asked to modify the solution to make it starvation-free.) Each reader thread alternates between sleeping and reading, as shown in Figure 6.15. When a reader wishes to read the database, it invokes the `acquireReadLock()` method; when it has finished reading, it calls `releaseReadLock()`. Each writer thread (Figure 6.16) performs similarly.

The methods called by each reader and writer thread are defined in the `ReadWriteLock` interface in Figure 6.17. The `Database` class in Figure 6.18

```
public interface ReadWriteLock
{
   public void acquireReadLock();
   public void acquireWriteLock();
   public void releaseReadLock();
   public void releaseWriteLock();
}
```

Figure 6.17 The interface for the readers–writers problem.

```
public class Database implements ReadWriteLock
{
   private int readerCount;
   private Semaphore mutex;
   private Semaphore db;

   public Database() {
      readerCount = 0;
      mutex = new Semaphore(1);
      db = new Semaphore(1);
   }

   public void acquireReadLock() {
      // Figure 6.19
   }

   public void releaseReadLock() {
      // Figure 6.19
   }

   public void acquireWriteLock() {
      // Figure 6.20
   }

   public void releaseWriteLock() {
      // Figure 6.20
   }
}
```

Figure 6.18 The database for the readers–writers problem.

implements this interface. The `readerCount` keeps track of the number of readers. The semaphore `mutex` is used to ensure mutual exclusion when `readerCount` is updated. The semaphore `db` functions as a mutual exclusion semaphore for the writers. It also is used by the readers to prevent writers from entering the database while the database is being read. The first reader performs an `acquire()` operation on `db`, thereby preventing any writers from entering the database. The final reader performs a `release()` operation on `db`. Note that, if a writer is active in the database and n readers are waiting, then one reader is queued on `db` and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `db.release()`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

Read–write locks are most useful in the following situations:

- In applications where it is easy to identify which threads only read shared data and which threads only write shared data.
- In applications that have more readers than writers. This is because read–write locks generally require more overhead to establish than semaphores

```
public void acquireReadLock() {
   mutex.acquire();

  /**
   * The first reader indicates that
   * the database is being read.
   */
   ++readerCount;
   if (readerCount == 1)
     db.acquire();

   mutex.release();
}

public void releaseReadLock() {
   mutex.acquire();

  /**
   * The last reader indicates that
   * the database is no longer being read.
   */
   --readerCount;
   if (readerCount == 0)
     db.release();

   mutex.release();
}
```

Figure 6.19 Methods called by readers.

or mutual exclusion locks, and the overhead for setting up a read–write lock is balanced by the increased concurrency of allowing multiple readers.

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 to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 6.21). When a philosopher thinks, she does

```
public void acquireWriteLock() {
  db.acquire();
}

public void releaseWriteLock() {
  db.release();
}
```

Figure 6.20 Methods called by writers.

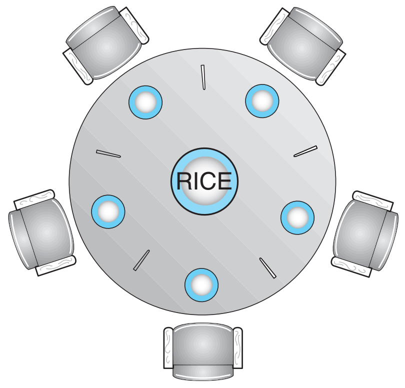

Figure 6.21 The situation of the dining philosophers.

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 the chopstick by executing an `acquire()` operation on that semaphore; she releases a chopstick by executing the `release()` operation on the appropriate semaphores. Thus, the shared data are

```
Semaphore chopStick[] = new Semaphore[5];

for(int i = 0; i < 5; i++)
   chopStick[i] = new Semaphore(1);
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 6.22.

Although this solution guarantees that no two neighboring philosophers are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating 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. These remedies prevent deadlock by placing restrictions on the philosophers:

```
while (true) {
    // get left chopstick
    chopStick[i].acquire();
    // get right chopstick
    chopStick[(i + 1) % 5].acquire();

    eating();

    // return left chopstick
    chopStick[i].release();
    // return right chopstick
    chopStick[(i + 1) % 5].release();

    thinking();
}
```

Figure 6.22 The structure of philosopher *i*.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (note that she must pick them up in a critical section).
- Use an asymmetric solution; for example, 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 `mutex.acquire()` before entering the critical section and `mutex.release()` afterward. If this sequence is not observed, two

```
dp.takeForks(i);
eat();
dp.returnForks(i);
```

It is easy to show that this solution ensures that no two neighboring philosophers are eating simultaneously and that no deadlocks will occur. However, it is possible for a philosopher to starve to death. We do not present a solution to that problem but rather ask you in the chapter-ending exercises to develop one.

6.8 Java Synchronization

Now that we have provided a grounding in synchronization theory, we can describe how Java synchronizes the activity of threads, allowing the programmer to develop generalized solutions to enforce mutual exclusion between threads. When an application ensures that data remain consistent even when accessed concurrently by multiple threads, the application is said to be **thread-safe**.

6.8.1 Bounded Buffer

In Chapter 3, we described a shared-memory solution to the bounded-buffer problem. This solution suffers from two disadvantages. First, both the producer and the consumer use busy-waiting loops if the buffer is either full or empty. Second, the variable `count`, which is shared by the producer and the consumer, may develop a race condition, as described in Section 6.1. This section addresses these and other problems while developing a solution using Java synchronization mechanisms.

6.8.1.1 Busy Waiting and Livelock

Busy waiting was introduced in Section 6.5.2, where we examined an implementation of the `acquire()` and `release()` semaphore operations. In that section, we described how a process could block itself as an alternative to busy waiting. One way to accomplish such blocking in Java is to have a thread call the `Thread.yield()` method. Recall from Section 5.7 that, when a thread invokes the `yield()` method, the thread stays in the runnable state but allows the JVM to select another runnable thread to run. The `yield()` method makes more effective use of the CPU than busy waiting does.

In this instance, however, using *either* busy waiting or yielding may lead to another problem, known as **livelock**. Livelock is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons. Deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another blocked thread in the set. Livelock occurs when a thread continuously attempts an action that fails.

Here is one scenario that could cause livelock. Recall that the JVM schedules threads using a priority-based algorithm, favoring high-priority threads over threads with lower priority. If the producer has a priority higher than that of the consumer and the buffer is full, the producer will enter the `while` loop

```
// Producers call this method
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE)
     Thread.yield();

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   ++count;
}


// Consumers call this method
public synchronized E remove() {
   E item;

   while (count == 0)
     Thread.yield();

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   --count;

   return item;
}
```

Figure 6.27 Synchronized `insert()` and `remove()` methods.

and either busy-wait or `yield()` to another runnable thread while waiting for `count` to be decremented to less than `BUFFER_SIZE`. As long as the consumer has a priority lower than that of the producer, it may never be scheduled by the JVM to run and therefore may never be able to consume an item and free up buffer space for the producer. In this situation, the producer is livelocked waiting for the consumer to free buffer space. We will see shortly that there is a better alternative than busy waiting or yielding while waiting for a desired event to occur.

6.8.1.2 Race Condition

In Section 6.1, we saw an example of the consequences of a race condition on the shared variable `count`. Figure 6.27 illustrates how Java's handling of concurrent access to shared data prevents race conditions.

In describing this situation, we introduce a new keyword: `synchronized`. Every object in Java has associated with it a single lock. An object's lock may be owned by a single thread. Ordinarily, when an object is being referenced (that is, when its methods are being invoked), the lock is ignored. When a method is declared to be `synchronized`, however, calling the method requires owning the lock for the object. 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`

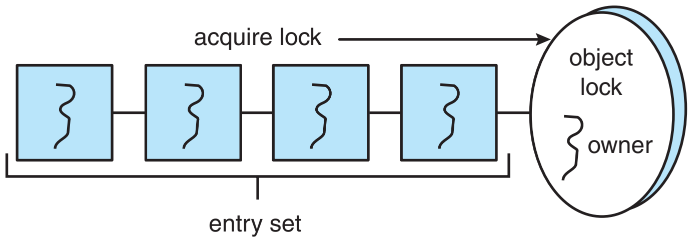

Figure 6.28 Entry set.

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. If the entry set for the lock is not empty when the lock is released, the JVM arbitrarily selects a thread from this set to be the owner of the lock. (When we say "arbitrarily," we mean that the specification does not require that threads in this set be organized in any particular order. However, in practice, most virtual machines order threads in the wait set according to a FIFO policy.) Figure 6.28 illustrates how the entry set operates.

If the producer calls the `insert()` method, as shown in Figure 6.27, and the lock for the object is available, the producer becomes the owner of the lock; it can then enter the method, where it can alter the value of `count` and other shared data. If the consumer attempts to call the `synchronized remove()` method while the producer owns the lock, the consumer will block because the lock is unavailable. When the producer exits the `insert()` method, it releases the lock. The consumer can now acquire the lock and enter the `remove()` method.

6.8.1.3 Deadlock

At first glance, this approach appears at least to solve the problem of having a race condition on the variable `count`. Because both the `insert()` method and the `remove()` method are declared `synchronized`, we have ensured that only one thread can be active in either of these methods at a time. However, lock ownership has led to another problem.

Assume that the buffer is full and the consumer is sleeping. If the producer calls the `insert()` method, it will be allowed to continue, because the lock is available. When the producer invokes the `insert()` method, it sees that the buffer is full and performs the `yield()` method. All the while, the producer still owns the lock for the object. When the consumer awakens and tries to call the `remove()` method (which would ultimately free up buffer space for the producer), it will block because it does not own the lock for the object. Thus, both the producer and the consumer are unable to proceed because (1) the producer is blocked waiting for the consumer to free space in the buffer and (2) the consumer is blocked waiting for the producer to release the lock.

By declaring each method as `synchronized`, we have prevented the race condition on the shared variables. However, the presence of the `yield()` loop has led to a possible deadlock.

```
// Producers call this method
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   ++count;

   notify();
}

// Consumers call this method
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   --count;

   notify();

   return item;
}
```

Figure 6.29 `insert()` and `remove()` methods using `wait()` and `notify()`.

6.8.1.4 Wait and Notify

Figure 6.29 addresses the `yield()` loop by introducing two new Java methods: `wait()` and `notify()`. In addition to having a lock, every object also has associated with it a **wait set** consisting of a set of threads. This wait set is initially empty. When a thread enters a `synchronized` method, it owns the lock for the object. However, this thread may determine that it is unable to continue because a certain condition has not been met. That will happen, for example, if the producer calls the `insert()` method and the buffer is full. The thread then will release the lock and wait until the condition that will allow it to continue is met, thus avoiding the previous deadlock situation.

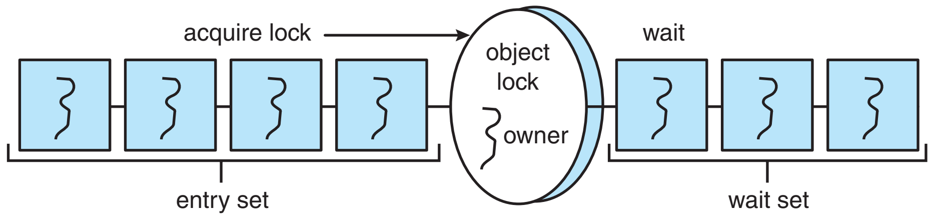

Figure 6.30 Entry and wait sets.

When a thread calls the `wait()` method, the following happens:

1. The thread releases the lock for the object.
2. The state of the thread is set to blocked.
3. The thread is placed in the wait set for the object.

Consider the example in Figure 6.29. If the producer calls the `insert()` method and sees that the buffer is full, it calls the `wait()` method. This call releases the lock, blocks the producer, and puts the producer in the wait set for the object. Because the producer has released the lock, the consumer ultimately enters the `remove()` method, where it frees space in the buffer for the producer. Figure 6.30 illustrates the entry and wait sets for a lock. (Note that `wait()` can result in an `InterruptedException` being thrown. We will cover this in Section 6.8.6.)

How does the consumer thread signal that the producer may now proceed? Ordinarily, when a thread exits a `synchronized` method, the departing thread releases only the lock associated with the object, possibly removing a thread from the entry set and giving it ownership of the lock. However, at the end of the `synchronized insert()` and `remove()` methods, we have a call to the method `notify()`. The call to `notify()`:

1. Picks an arbitrary thread `T` from the list of threads in the wait set
2. Moves `T` from the wait set to the entry set
3. Sets the state of `T` from blocked to runnable

`T` is now eligible to compete for the lock with the other threads. Once `T` has regained control of the lock, it returns from calling `wait()`, where it may check the value of `count` again.

Next, we describe the `wait()` and `notify()` methods in terms of the program shown in Figure 6.29. We assume that the buffer is full and the lock for the object is available.

- The producer calls the `insert()` method, sees that the lock is available, and enters the method. Once in the method, the producer determines that the buffer is full and calls `wait()`. The call to `wait()` releases the lock for the object, sets the state of the producer to blocked, and puts the producer in the wait set for the object.

- The consumer ultimately calls and enters the `remove()` method, as the lock for the object is now available. The consumer removes an item from the buffer and calls `notify()`. Note that the consumer still owns the lock for the object.
- The call to `notify()` removes the producer from the wait set for the object, moves the producer to the entry set, and sets the producer's state to runnable.
- The consumer exits the `remove()` method. Exiting this method releases the lock for the object.
- The producer tries to reacquire the lock and is successful. It resumes execution from the call to `wait()`. The producer tests the `while` loop, determines that room is available in the buffer, and proceeds with the remainder of the `insert()` method. If no thread is in the wait set for the object, the call to `notify()` is ignored. When the producer exits the method, it releases the lock for the object.

The `BoundedBuffer` class shown in Figure 6.31 represents the complete solution to the bounded-buffer problem using Java synchronization. This class may be substituted for the `BoundedBuffer` class used in the semaphore-based solution to this problem in Section 6.6.1.

```
public class BoundedBuffer<E> implements Buffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
     // buffer is initially empty
     count = 0;
     in = 0;
     out = 0;
     buffer = (E[]) new Object[BUFFER_SIZE];
   }

   public synchronized void insert(E item) {
      // Figure 6.29
   }

   public synchronized E remove() {
      // Figure 6.29
   }
}
```

Figure 6.31 Bounded buffer.

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public synchronized void doWork(int myNumber) {
   while (turn != myNumber) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   // Do some work for awhile . . .

  /**
   * Finished working. Now indicate to the
   * next waiting thread that it is their
   * turn to do some work.
   */
   turn = (turn + 1) % 5;

   notify();
}
```

Figure 6.32 `doWork()` method.

6.8.2 Multiple Notifications

As described in Section 6.8.1.4, the call to `notify()` arbitrarily selects a thread from the list of threads in the wait set for an object. This approach works fine when only one thread is in the wait set, but consider what can happen when there are multiple threads in the wait set and more than one condition for which to wait. It is possible that a thread whose condition has not yet been met will be the thread that receives the notification.

Suppose, for example, that there are five threads $\{T0, T1, T2, T3, T4\}$ and a shared variable `turn` indicating which thread's turn it is. When a thread wishes to do work, it calls the `doWork()` method in Figure 6.32. Only the thread whose number matches the value of `turn` can proceed; all other threads must wait their turn.

Assume the following:

- `turn = 3`.
- $T1$, $T2$, and $T4$ are in the wait set for the object.
- $T3$ is currently in the `doWork()` method.

When thread $T3$ is done, it sets `turn` to 4 (indicating that it is $T4$'s turn) and calls `notify()`. The call to `notify()` arbitrarily picks a thread from the wait set. If $T2$ receives the notification, it resumes execution from the call to `wait()` and tests the condition in the `while` loop. $T2$ sees that this is not its turn, so it calls `wait()` again. Ultimately, $T3$ and $T0$ will call `doWork()` and will also invoke the `wait()` method, since it is the turn for neither $T3$ nor $T0$. Now, all five threads are blocked in the wait set for the object. Thus, we have another deadlock to handle.

Because the call to `notify()` arbitrarily picks a single thread from the wait set, the developer has no control over which thread is chosen. Fortunately, Java provides a mechanism that allows all threads in the wait set to be notified. The `notifyAll()` method is similar to `notify()`, except that *every* waiting thread is removed from the wait set and placed in the entry set. If the call to `notify()` in `doWork()` is replaced with a call to `notifyAll()`, when $T3$ finishes and sets `turn` to 4, it calls `notifyAll()`. This call has the effect of removing $T1$, $T2$, and $T4$ from the wait set. The three threads then compete for the object's lock once again. Ultimately, $T1$ and $T2$ call `wait()`, and only $T4$ proceeds with the `doWork()` method.

In sum, the `notifyAll()` method is a mechanism that wakes up all waiting threads and lets the threads decide among themselves which of them should run next. In general, `notifyAll()` is a more expensive operation than `notify()` because it wakes up all threads, but it is regarded as a more conservative strategy appropriate for situations in which multiple threads may be in the wait set for an object.

```
public class Database implements ReadWriteLock
{
   private int readerCount;
   private boolean dbWriting;

   public Database() {
     readerCount = 0;
     dbWriting = false;
   }

   public synchronized void acquireReadLock() {
     // Figure 6.34
   }

   public synchronized void releaseReadLock() {
     // Figure 6.34
   }

   public synchronized void acquireWriteLock() {
     // Figure 6.35
   }

   public synchronized void releaseWriteLock() {
     // Figure 6.35
   }
}
```

Figure 6.33 Solution to the readers–writers problem using Java synchronization.

```
public synchronized void acquireReadLock() {
    while (dbWriting == true) {
       try {
          wait();
       }
       catch(InterruptedException e) { }
    }

    ++readerCount;
}

public synchronized void releaseReadLock() {
    --readerCount;

   /**
    * The last reader indicates that
    * the database is no longer being read.
    */
    if (readerCount == 0)
       notify();
}
```

Figure 6.34 Methods called by readers.

In the following section, we look at a Java-based solution to the readers–writers problem that requires the use of both `notify()` and `notifyAll()`.

6.8.3 A Java-Based Solution to the Readers–Writers Problem

We can now provide a solution to the first readers–writers problem by using Java synchronization. The methods called by each reader and writer thread are defined in the `Database` class in Figure 6.33, which implements the `ReadWriteLock` interface shown in Figure 6.17. The `readerCount` keeps track of the number of readers; a value > 0 indicates that the database is currently being read. `dbWriting` is a boolean variable indicating whether the database is currently being accessed by a writer. `acquireReadLock()`, `releaseReadLock()`, `acquireWriteLock()`, and `releaseWriteLock()` are all declared as `synchronized` to ensure mutual exclusion to the shared variables.

When a writer wishes to begin writing, it first checks whether the database is currently being either read or written. If the database is being read or written, the writer enters the wait set for the object. Otherwise, it sets `dbWriting` to `true`. When a writer is finished, it sets `dbWriting` to `false`. When a reader invokes `acquireReadLock()`, it first checks whether the database is currently being written. If the database is unavailable, the reader enters the wait set for the object; otherwise, it increments `readerCount`. The final reader calling `releaseReadLock()` invokes `notify()`, thereby notifying a waiting writer. When a writer invokes `releaseWriteLock()`, however, it calls the `notifyAll()` method rather than `notify()`. Consider the effect on readers. If several readers wish to read the database while it is being written, and the

```
public synchronized void acquireWriteLock() {
   while (readerCount > 0 || dbWriting == true) {
      try {
         wait();
      }
      catch(InterruptedException e) { }
   }

  /**
   * Once there are no readers or a writer,
   * indicate that the database is being written.
   */
   dbWriting = true;
}

public synchronized void releaseWriteLock() {
   dbWriting = false;

   notifyAll();
}
```

Figure 6.35 Methods called by writers.

writer invokes `notify()` once it has finished writing, only one reader will receive the notification. Other readers will remain in the wait set even though the database is available for reading. By invoking `notifyAll()`, a departing writer is ensured of notifying all waiting readers.

6.8.4 Block Synchronization

The amount of time between when a lock is acquired and when it is released is defined as the **scope** of the lock. Java also allows blocks of code to be declared as `synchronized`, because a `synchronized` method that has only a small percentage of its code manipulating shared data may yield a scope that is too large. In such an instance, it may be better to synchronize only the block of code that manipulates shared data than to synchronize the entire method. Such a design results in a smaller lock scope. Thus, in addition to declaring `synchronized` methods, Java also allows block synchronization, as illustrated in Figure 6.36. Access to the `criticalSection()` method in Figure 6.36 requires ownership of the lock for the `mutexLock` object.

We can also use the `wait()` and `notify()` methods in a synchronized block. The only difference is that they must be invoked with the same object that is being used for synchronization. This approach is shown in Figure 6.37.

6.8.5 Synchronization Rules

The `synchronized` keyword is a straightforward construct, but it is important to know a few rules about its behavior.

```
Object mutexLock = new Object();
. . .
public void someMethod() {
   nonCriticalSection();

   synchronized(mutexLock) {
      criticalSection();
   }

   remainderSection();
}
```

Figure 6.36 Block synchronization.

1. A thread that owns the lock for an object can enter another `synchronized` method (or block) for the same object. This is known as a **recursive** or **reentrant** lock.
2. A thread can nest `synchronized` method invocations for different objects. Thus, a thread can simultaneously own the lock for several different objects.
3. If a method is not declared `synchronized`, then it can be invoked regardless of lock ownership, even while another `synchronized` method for the same object is executing.
4. If the wait set for an object is empty, then a call to `notify()` or `notifyAll()` has no effect.
5. `wait()`, `notify()`, and `notifyAll()` may only be invoked from `synchronized` methods or blocks; otherwise, an `IllegalMonitorStateException` is thrown.

It is also possible to declare `static` methods as `synchronized`. This is because, along with the locks that are associated with object instances, there is a single **class lock** associated with each class. Thus, for a given class, there

```
Object mutexLock = new Object();
. . .
synchronized(mutexLock) {
    try {
       mutexLock.wait();
    }
    catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
    mutexLock.notify();
}
```

Figure 6.37 Block synchronization using `wait()` and `notify()`.

can be several object locks, one per object instance. However, there is only one class lock.

In addition to using the class lock to declare `static` methods as `synchronized`, we can use it in a `synchronized` block by placing "*class name*`.class`" within the `synchronized` statement. For example, if we wished to use a `synchronized` block with the class lock for the `SomeObject` class, we would use the following:

```
synchronized(SomeObject.class) {
   /**
    * synchronized block of code
    */
}
```

6.8.6 Handling `InterruptedException`

Note that invoking the `wait()` method requires placing it in a `try-catch` block, as `wait()` may throw an `InterruptedException`. Recall from Chapter 4 that the `interrupt()` method is the preferred technique for interrupting a thread in Java. When `interrupt()` is invoked on a thread, the **interruption status** of that thread is set. A thread can check its interruption status using the `isInterrupted()` method, which returns `true` if its interruption status is set.

The `wait()` method also checks the interruption status of a thread. If it is set, `wait()` will throw an `InterruptedException`. This allows interruption of a thread that is blocked in the wait set. (It should also be noted that once an `InterruptedException` is thrown, the interrupted status of the thread is cleared.) For code clarity and simplicity, we choose to ignore this exception in our code examples. That is, all calls to `wait()` appear as:

```
try {
    wait();
}
catch (InterruptedException ie) { /* ignore */ }
```

However, if we choose to handle `InterruptedException`, we permit the interruption of a thread blocked in a wait set. Doing so allows more robust multithreaded applications, as it provides a mechanism for interrupting a thread that is blocked trying to acquire a mutual exclusion lock. One strategy is to allow the `InterruptedException` to propagate. That is, in methods where `wait()` is invoked, we first remove the `try-catch` blocks when calling `wait()` and declare such methods as throwing `InterruptedException`. By doing this, we are allowing the `InterruptedException` to propagate from the method where `wait()` is being invoked. However, allowing this exception to propagate requires placing calls to such methods within `try-catch (InterruptedException)` blocks.

6.8.7 Concurrency Features in Java

Prior to Java 1.5, the only concurrency features available in Java were the `synchronized`, `wait()`, and `notify()` commands, which are based on single locks for each object. Java 1.5 introduced a rich API consisting of several concurrency features, including various mechanisms for synchronizing concurrent threads. In this section, we cover (1) reentrant locks, (2) semaphores, and (3) condition variables available in the `java.util.concurrent` and `java.util.concurrent.locks` packages. Readers interested in the additional features of these packages are encouraged to consult the Java API.

6.8.7.1 Reentrant Locks

Perhaps the simplest locking mechanism available in the API is the `ReentrantLock`. In many ways, a `ReentrantLock` acts like the `synchronized` statement described in Section 6.8.1.2: a `ReentrantLock` is owned by a single thread and is used to provide mutually exclusive access to a shared resource. However, the `ReentrantLock` provides several additional features, such as setting a *fairness* parameter, which favors granting the lock to the longest-waiting thread. (Recall from Section 6.8.1.2 that the specification for the JVM does not indicate that threads in the wait set for an object lock are to be ordered in any specific fashion.)

A thread acquires a `ReentrantLock` lock by invoking its `lock()` method. If the lock is available—or if the thread invoking `lock()` already owns it, which is why it is termed *reentrant*—`lock()` assigns the invoking thread lock ownership and returns control. If the lock is unavailable, the invoking thread blocks until it is ultimately assigned the lock when its owner invokes `unlock()`. `ReentrantLock` implements the `Lock` interface; its usage is as follows:

```
Lock key = new ReentrantLock();

key.lock();
try {
   // critical section
}
finally {
   key.unlock();
}
```

The programming idiom of using `try` and `finally` requires a bit of explanation. If the lock is acquired via the `lock()` method, it is important that the lock be similarly released. By enclosing `unlock()` in a `finally` clause, we ensure that the lock is released once the critical section completes or if an exception occurs within the `try` block. Notice that we do not place the call to `lock()` within the `try` clause, as `lock()` does not throw any checked exceptions. Consider what happens if we place `lock()` within the `try` clause and an unchecked exception occurs when `lock()` is invoked (such as `OutofMemoryError`): The `finally` clause triggers the call to `unlock()`, which then throws the unchecked `IllegalMonitorStateException`, as the lock was never acquired. This `IllegalMonitorStateException` replaces the unchecked exception that occurred when `lock()` was invoked, thereby obscuring the reason why the program initially failed.

6.8.7.2 Semaphores

The Java 5 API also provides a counting semaphore, as described in Section 6.5. The constructor for the semaphore appears as

```
Semaphore(int value);
```

where value specifies the initial value of the semaphore (a negative value is allowed). The acquire() method throws an InterruptedException if the acquiring thread is interrupted (Section 6.8.6). The following example illustrates using a semaphore for mutual exclusion:

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   // critical section
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Notice that we place the call to release() in the finally clause to ensure that the semaphore is released.

6.8.7.3 Condition Variables

The last utility we cover in the Java API is the condition variable. Just as the ReentrantLock (Section 6.8.7.1) is similar to Java's synchronized statement, condition variables provide functionality similar to the wait(), notify(), and notifyAll() methods. Therefore, to provide mutual exclusion to both, a condition variable must be associated with a reentrant lock.

We create a condition variable by first creating a ReentrantLock and invoking its newCondition() method, which returns a Condition object representing the condition variable for the associated ReentrantLock. This is illustrated in the following statements:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

Once the condition variable has been obtained, we can invoke its await() and signal() methods, which function in the same way as the wait() and signal() commands described in Section 6.7.

As mentioned, reentrant locks and condition variables in the Java API function similarly to the synchronized, wait(), and notify() statements. However, one advantage to using the features available in the API is they often provide more flexibility and control than their synchronized/wait()/notify() counterparts. Another distinction concerns Java's locking mechanism, in which each object has its own lock. In many ways, this lock acts as a monitor. Every Java object thus has an associated monitor, and a thread can acquire an object's monitor by entering a synchronized method or block.

Let's look more closely at this distinction. Recall that, with monitors as described in Section 6.7, the wait() and signal() operations can be applied to *named* condition variables, allowing a thread to wait for a specific condition or to be notified when a specific condition has been met. At the language level, Java does not provide support for named condition variables. Each Java monitor is associated with just one unnamed condition variable, and the `wait()`, `notify()`, and `notifyAll()` operations apply only to this single condition variable. When a Java thread is awakened via `notify()` or `notifyAll()`, it receives no information as to why it was awakened. It is up to the reactivated thread to check for itself whether the condition for which it was waiting has been met. The `doWork()` method shown in Figure 6.32 highlights this issue; `notifyAll()` must be invoked to awaken all waiting threads, and—once awake—each thread must check for itself if the condition it has been waiting for has been met (that is, if it is that thread's turn).

We further illustrate this distinction by rewriting the `doWork()` method in Figure 6.32 using condition variables. We first create a `ReentrantLock` and five condition variables (representing the conditions the threads are waiting for) to signal the thread whose turn is next. This is shown below:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
   condVars[i] = lock.newCondition();
```

The modified `doWork()` method is shown in Figure 6.38. Notice that `doWork()` is no longer declared as `synchronized`, since the `ReentrantLock` provides mutual exclusion. When a thread invokes `await()` on the condition variable, it releases the associated `ReentrantLock`, allowing another thread to acquire the mutual exclusion lock. Similarly, when `signal()` is invoked, only the condition variable is signaled; the lock is released by invoking `unlock()`.

6.9 Synchronization Examples

We next describe the synchronization mechanisms provided by the Solaris, Windows XP, 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 for 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.

6.9.1 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