Synchronization Examples

In Chapter 6, we presented the critical-section problem and focused on how race conditions can occur when multiple concurrent processes share data. We went on to examine several tools that address the critical-section problem by preventing race conditions from occurring. These tools ranged from low-level hardware solutions (such as memory barriers and the compare-and-swap operation) to increasingly higher-level tools (from mutex locks to semaphores to monitors). We also discussed various challenges in designing applications that are free from race conditions, including liveness hazards such as deadlocks. In this chapter, we apply the tools presented in Chapter 6 to several classic synchronization problems. We also explore the synchronization mechanisms used by the Linux, UNIX, and Windows operating systems, and we describe API details for both Java and POSIX systems.

CHAPTER OBJECTIVES

- Explain the bounded-buffer, readers–writers, and dining–philosophers synchronization problems.
- Describe specific tools used by Linux and Windows to solve process synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems.
- Design and develop solutions to process synchronization problems using POSIX and Java APIs.

7.1 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, since that is the

```
while (true) {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
}
```

Figure 7.1 The structure of the producer process.

traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

7.1.1 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.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of `n` buffers, each capable of holding one item. The `mutex` binary 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 7.1, and the code for the consumer process is shown in Figure 7.2. 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.

7.1.2 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, read and write) the database. We distinguish

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

Figure 7.2 The structure of the consumer process.

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. See 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 rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The binary semaphores `mutex` and `rw_mutex` are initialized to 1; `read_count` is a counting semaphore initialized to 0. The semaphore `rw_mutex`

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `read_count` is updated. The `read_count` variable keeps track of how many processes are currently reading the object. The semaphore `rw_mutex` 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 that enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 7.3; the code for a reader process is shown in Figure 7.4. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `rw_mutex`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(rw_mutex)`, 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 requires specifying the mode of the lock: either *read* or *write* access. When a

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

Figure 7.4 The structure of a reader process.

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.

7.1.3 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 7.5). 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 the chopsticks. When she is finished eating, she puts down both 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

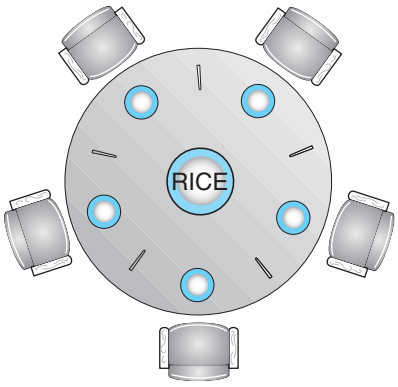

Figure 7.5 The situation of the dining philosophers.

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

Figure 7.6 The structure of philosopher *i*.

to allocate several resources among several processes in a deadlock-free and starvation-free manner.

7.1.3.1 Semaphore Solution

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 7.6.

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 at the same time 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 the following:

- 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 (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered 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.

7.1.3.2 Monitor Solution

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];
```

This allows philosopher *i* to 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 7.7. 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 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. As we already noted, however, 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.

7.2 Synchronization within the Kernel

We next describe the synchronization mechanisms provided by the Windows and Linux operating systems. These two operating systems provide good examples of different approaches to synchronizing the kernel, and as you will

```
monitor DiningPhilosophers
{
   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 7.7 A monitor solution to the dining-philosophers problem.

see, the synchronization mechanisms available in these systems differ in subtle yet significant ways.

7.2.1 Synchronization in Windows

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 single-processor 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, although 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 mutex locks, 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 6.6. **Events** are similar to condition variables; that is, they may 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. An object in a **signaled state** is available, and a thread will not block when acquiring the object. An object in a **nonsignaled state** 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 7.8.

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 each thread 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 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.

A **critical-section object** is a user-mode mutex that can often be acquired and released without kernel intervention. On a multiprocessor system, a critical-section object first uses a spinlock while waiting for the other thread to release the object. If it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU. Critical-section objects are particularly efficient because the kernel mutex is allocated only when there is contention for the object. In practice, there is very little contention, so the savings are significant.

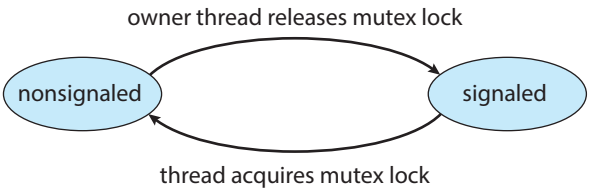

Figure 7.8 Mutex dispatcher object.

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

7.2.2 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.

Linux provides several different mechanisms for synchronization in the kernel. As most computer architectures provide instructions for atomic versions of simple math operations, the simplest synchronization technique within the Linux kernel is an atomic integer, which is represented using the opaque data type `atomic_t`. As the name implies, all math operations using atomic integers are performed without interruption. To illustrate, consider a program that consists of an atomic integer `counter` and an integer `value`.

```
atomic_t counter;
int value;
```

The following code illustrates the effect of performing various atomic operations:

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Atomic integers are particularly efficient in situations where an integer variable—such as a counter—needs to be updated, since atomic operations do not require the overhead of locking mechanisms. However, their use is limited to these sorts of scenarios. In situations where there are several variables contributing to a possible race condition, more sophisticated locking tools must be used.

Mutex locks are available in Linux for protecting critical sections within the kernel. Here, a task must invoke the `mutex_lock()` function prior to entering a critical section and the `mutex_unlock()` function after exiting the critical section. If the mutex lock is unavailable, a task calling `mutex_lock()` is put into a sleep state and is awakened when the lock's owner invokes `mutex_unlock()`.

Linux also 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, such as embedded systems with only a single processing core, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on systems with a single processing core, 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

In the Linux kernel, both spinlocks and mutex locks are *nonrecursive*, which means that if a thread has acquired one of these locks, it cannot acquire the same lock a second time without first releasing the lock. Otherwise, the second attempt at acquiring the lock will block.

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. The kernel is not preemptible, however, if a task running in the kernel 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 in the kernel is greater than 0, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is 0, 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 or mutex locks are appropriate for use.

7.3 POSIX Synchronization

The synchronization methods discussed in the preceding section pertain to synchronization within the kernel and are therefore available only to kernel developers. In contrast, the POSIX API is available for programmers at the user level and is not part of any particular operating-system kernel. (Of course, it must ultimately be implemented using tools provided by the host operating system.)

In this section, we cover mutex locks, semaphores, and condition variables that are available in the Pthreads and POSIX APIs. These APIs are widely used for thread creation and synchronization by developers on UNIX, Linux, and macOS systems.

7.3.1 POSIX Mutex Locks

Mutex locks represent 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. Pthreads uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init()` function. The first parameter is a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes. This is illustrated below:

```
#include <pthread.h>

pthread_mutex_t mutex;

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

The mutex is acquired and released with the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`. The following code illustrates protecting a critical section with mutex locks:

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

/* critical section */

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

All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

7.3.2 POSIX Semaphores

Many systems that implement Pthreads also provide semaphores, although semaphores are not part of the POSIX standard and instead belong to the POSIX SEM extension. POSIX specifies two types of semaphores—**named** and **unnamed**. Fundamentally, the two are quite similar, but they differ in terms of how they are created and shared between processes. Because both techniques are common, we discuss both here. Beginning with Version 2.6 of the kernel, Linux systems provide support for both named and unnamed semaphores.

7.3.2.1 POSIX Named Semaphores

The function `sem_open()` is used to create and open a POSIX named sempahore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

In this instance, we are naming the semaphore `SEM`. The `O_CREAT` flag indicates that the semaphore will be created if it does not already exist. Additionally, the semaphore has read and write access for other processes (via the parameter `0666`) and is initialized to 1.

The advantage of named semaphores is that multiple unrelated processes can easily use a common semaphore as a synchronization mechanism by

simply referring to the semaphore's name. In the example above, once the semaphore SEM has been created, subsequent calls to sem_open() (with the same parameters) by other processes return a descriptor to the existing semaphore.

In Section 6.6, we described the classic wait() and signal() semaphore operations. POSIX declares these operations sem_wait() and sem_post(), respectively. The following code sample illustrates protecting a critical section using the named semaphore created above:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Both Linux and macOS systems provide POSIX named semaphores.

7.3.2.2 POSIX Unnamed Semaphores

An unnamed semaphore is created and initialized using the sem_init() function, which is passed three parameters:

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

and is illusrated in the following programming example:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

In this example, by passing the flag 0, we are indicating that this semaphore can be shared only by threads belonging to the process that created the semaphore. (If we supplied a nonzero value, we could allow the semaphore to be shared between separate processes by placing it in a region of shared memory.) In addition, we initialize the semaphore to the value 1.

POSIX unnamed semaphores use the same sem_wait() and sem_post() operations as named semaphores. The following code sample illustrates protecting a critical section using the unnamed semaphore created above:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

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

Just like mutex locks, all semaphore functions return 0 when successful and nonzero when an error condition occurs.

7.3.3 POSIX Condition Variables

Condition variables in Pthreads behave similarly to those described in Section 6.7. However, in that section, condition variables are used within the context of a monitor, which provides a locking mechanism to ensure data integrity. Since Pthreads is typically used in C programs—and since C does not have a monitor— we accomplish locking by associating a condition variable with a mutex lock.

Condition variables in Pthreads use the `pthread_cond_t` data type and are initialized using the `pthread_cond_init()` function. The following code creates and initializes a condition variable as well as its associated mutex lock:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

The `pthread_cond_wait()` function is used for waiting on a condition variable. The following code illustrates how a thread can wait for the condition `a == b` to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

The mutex lock associated with the condition variable must be locked before the `pthread_cond_wait()` function is called, since it is used to protect the data in the conditional clause from a possible race condition. Once this lock is acquired, the thread can check the condition. If the condition is not true, the thread then invokes `pthread_cond_wait()`, passing the mutex lock and the condition variable as parameters. Calling `pthread_cond_wait()` releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true. (To protect against program errors, it is important to place the conditional clause within a loop so that the condition is rechecked after being signaled.)

A thread that modifies the shared data can invoke the `pthread_cond_signal()` function, thereby signaling one thread waiting on the condition variable. This is illustrated below:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

It is important to note that the call to `pthread_cond_signal()` does not release the mutex lock. It is the subsequent call to `pthread_mutex_unlock()` that releases the mutex. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to `pthread_cond_wait()`.

We provide several programming problems and projects at the end of this chapter that use Pthreads mutex locks and condition variables, as well as POSIX semaphores.

7.4 Synchronization in Java

The Java language and its API have provided rich support for thread synchronization since the origins of the language. In this section, we first cover Java monitors, Java's original synchronization mechanism. We then cover three additional mechanisms that were introduced in Release 1.5: reentrant locks, semaphores, and condition variables. We include these because they represent the most common locking and synchronization mechanisms. However, the Java API provides many features that we do not cover in this text—for example, support for atomic variables and the CAS instruction—and we encourage interested readers to consult the bibliography for more information.

7.4.1 Java Monitors

Java provides a monitor-like concurrency mechanism for thread synchronization. We illustrate this mechanism with the `BoundedBuffer` class (Figure 7.9), which implements a solution to the bounded-buffer problem wherein the producer and consumer invoke the `insert()` and `remove()` methods, respectively.

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, such as with the `insert()` and `remove()` methods in the `BoundedBuffer` class.

Invoking a `synchronized` method requires owning the lock on an object instance of `BoundedBuffer`. 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. If the entry set for the lock is not empty when the lock is released, the JVM

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

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

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

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

Figure 7.9 Bounded buffer using Java synchronization.

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 entry set according to a FIFO policy.) Figure 7.10 illustrates how the entry set operates.

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

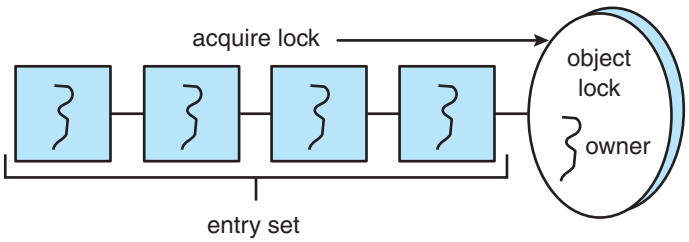

Figure 7.10 Entry set for a lock.

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. 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 below. Only the access to the critical-section code requires ownership of the object lock for the `this` object.

```
public void someMethod() {
   /* non-critical section */

   synchronized(this) {
      /* critical section */
   }

   /* remainder section */
}
```

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.

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 7.11. 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 7.12 illustrates the entry and wait sets for a lock. (Note that although `wait()` can throw an `InterruptedException`, we choose to ignore it for code clarity and simplicity.)

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 `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

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

   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 ie) { }
   }

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

   notify();

   return item;
}
```

Figure 7.11 `insert()` and `remove()` methods using `wait()` and `notify()`.

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. (Again, the selection of an *arbitrary* thread is according to the Java specification; in practice, most Java virtual machines order threads in the wait set according to a FIFO policy.)

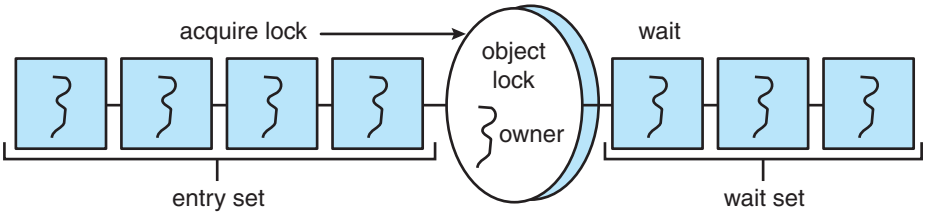

Figure 7.12 Entry and wait sets.

Next, we describe the `wait()` and `notify()` methods in terms of the methods shown in Figure 7.11. 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 `synchronized`, `wait()`, and `notify()` mechanisms have been part of Java since its origins. However, later revisions of the Java API introduced much more flexible and robust locking mechanisms, some of which we examine in the following sections.

7.4.2 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 7.4.1: 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

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; it is used 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.

Whereas a `ReentrantLock` provides mutual exclusion, it may be too conservative a strategy if multiple threads only read, but do not write, shared data. (We described this scenario in Section 7.1.2.) To address this need, the Java API also provides a `ReentrantReadWriteLock`, which is a lock that allows multiple concurrent readers but only one writer.

7.4.3 Semaphores

The Java API also provides a counting semaphore, as described in Section 6.6. 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. 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.

7.4.4 Condition Variables

The last utility we cover in the Java API is the condition variable. Just as the `ReentrantLock` is similar to Java's `synchronized` statement, condition variables provide functionality similar to the `wait()` and `notify()` methods. Therefore, to provide mutual exclusion, 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.

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()` and `notify()` operations described in Section 7.4.1 apply only to this single condition variable. When a Java thread is awakened via `notify()`, 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. Condition variables remedy this by allowing a specific thread to be notified.

We illustrate with the following example: Suppose we have five threads, numbered 0 through 4, 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 7.13, passing its thread number. Only the thread whose value of `threadNumber` matches the value of `turn` can proceed; other threads must wait their turn.

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled.
       */
      if (threadNumber != turn)
        condVars[threadNumber].await();

      /**
       * Do some work for awhile ...
       */

      /**
       * Now signal to the next thread.
       */
      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```

Figure 7.13 Example using Java condition variables.

We also must 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();
```

When a thread enters `doWork()`, it invokes the `await()` method on its associated condition variable if its `threadNumber` is not equal to `turn`, only to resume when it is signaled by another thread. After a thread has completed its work, it signals the condition variable associated with the thread whose turn follows.

It is important to note that `doWork()` does not need to be declared `synchronized`, as 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()`.

7.5 Alternative Approaches

With the emergence of multicore systems has come increased pressure to develop concurrent applications that take advantage of multiple processing cores. However, concurrent applications present an increased risk of race conditions and liveness hazards such as deadlock. Traditionally, techniques such as mutex locks, semaphores, and monitors have been used to address these issues, but as the number of processing cores increases, it becomes increasingly difficult to design multithreaded applications that are free from race conditions and deadlock. In this section, we explore various features provided in both programming languages and hardware that support the design of thread-safe concurrent applications.

7.5.1 Transactional Memory

Quite often in computer science, ideas from one area of study can be used to solve problems in other areas. The concept of **transactional memory** originated in database theory, for example, yet it provides a strategy for process synchronization. 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 mutex locks (or semaphores) such as the following:

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

However, using synchronization mechanisms such as mutex locks and semaphores involves many potential problems, including deadlock. Additionally, as the number of threads increases, traditional locking doesn't scale as well, because the level of contention among threads for lock ownership becomes very high.

As an alternative to traditional locking 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() function as follows:

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

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, because no locks are involved, deadlock is not possible. Furthermore, a transactional memory 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 and parallel programming have prompted a significant amount of research in this area on the part of both academics and commercial software and hardware vendors.

7.5.2 OpenMP

In Section 4.5.2, we provided an overview of OpenMP and its support of parallel programming in a shared-memory environment. Recall that OpenMP includes a set of compiler directives and an API. Any code following the compiler directive `#pragma omp parallel` is identified as a parallel region and is performed by a number of threads equal to the number of processing cores in the system. The advantage of OpenMP (and similar tools) is that thread creation and management are handled by the OpenMP library and are not the responsibility of application developers.

Along with its `#pragma omp parallel` compiler directive, OpenMP provides the compiler directive `#pragma omp critical`, which specifies the code region following the directive as a critical section in which only one thread may be active at a time. In this way, OpenMP provides support for ensuring that threads do not generate race conditions.

As an example of the use of the critical-section compiler directive, first assume that the shared variable `counter` can be modified in the `update()` function as follows:

```
void update(int value)
{
   counter += value;
}
```

If the `update()` function can be part of—or invoked from—a parallel region, a race condition is possible on the variable `counter`.

The critical-section compiler directive can be used to remedy this race condition and is coded as follows:

```
void update(int value)
{
   #pragma omp critical
   {
      counter += value;
   }
}
```

The critical-section compiler directive behaves much like a binary semaphore or mutex lock, ensuring that only one thread at a time is active in the critical section. If a thread attempts to enter a critical section when another thread is currently active in that section (that is, ***owns*** the section), the calling thread is blocked until the owner thread exits. If multiple critical sections must be used, each critical section can be assigned a separate name, and a rule can specify that no more than one thread may be active in a critical section of the same name simultaneously.

An advantage of using the critical-section compiler directive in OpenMP is that it is generally considered easier to use than standard mutex locks. However, a disadvantage is that application developers must still identify possible race conditions and adequately protect shared data using the compiler directive. Additionally, because the critical-section compiler directive behaves much like a mutex lock, deadlock is still possible when two or more critical sections are identified.

7.5.3 Functional Programming Languages

Most well-known programming languages—such as C, C++, Java, and C#—are known as **imperative** (or **procedural**) languages. Imperative languages are used for implementing algorithms that are state-based. In these languages, the flow of the algorithm is crucial to its correct operation, and state is represented with variables and other data structures. Of course, program state is mutable, as variables may be assigned different values over time.

With the current emphasis on concurrent and parallel programming for multicore systems, there has been greater focus on **functional** programming languages, which follow a programming paradigm much different from that offered by imperative languages. The fundamental difference between imperative and functional languages is that functional languages do not maintain state. That is, once a variable has been defined and assigned a value, its value

is immutable—it cannot change. Because functional languages disallow mutable state, they need not be concerned with issues such as race conditions and deadlocks. Essentially, most of the problems addressed in this chapter are nonexistent in functional languages.

Several functional languages are presently in use, and we briefly mention two of them here: Erlang and Scala. The Erlang language has gained significant attention because of its support for concurrency and the ease with which it can be used to develop applications that run on parallel systems. Scala is a functional language that is also object-oriented. In fact, much of the syntax of Scala is similar to the popular object-oriented languages Java and C#. Readers interested in Erlang and Scala, and in further details about functional languages in general, are encouraged to consult the bibliography at the end of this chapter for additional references.

7.6 Summary

- Classic problems of process synchronization include the bounded-buffer, readers–writers, and dining-philosophers problems. Solutions to these problems can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, monitors, and condition variables.
- Windows uses dispatcher objects as well as events to implement process synchronization tools.
- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.
- The POSIX API provides mutex locks, semaphores, and condition variables. POSIX provides two forms of semaphores: named and unnamed. Several unrelated processes can easily access the same named semaphore by simply referring to its name. Unnamed semaphores cannot be shared as easily, and require placing the semaphore in a region of shared memory.
- Java has a rich library and API for synchronization. Available tools include monitors (which are provided at the language level) as well as reentrant locks, semaphores, and condition variables (which are supported by the API).
- Alternative approaches to solving the critical-section problem include transactional memory, OpenMP, and functional languages. Functional languages are particularly intriguing, as they offer a different programming paradigm from procedural languages. Unlike procedural languages, functional languages do not maintain state and therefore are generally immune from race conditions and critical sections.

Practice Exercises

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

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

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

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

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

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

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

Further Reading

Details of Windows synchronization can be found in [Solomon and Russinovich (2000)]. [Love (2010)] describes synchronization in the Linux kernel. [Hart (2005)] describes thread synchronization using Windows. [Breshears (2009)] and [Pacheco (2011)] provide detailed coverage of synchronization issues in relation to parallel programming. Details on using OpenMP can be found at http://openmp.org. Both [Oaks (2014)] and [Goetz et al. (2006)] contrast traditional synchronization and CAS-based strategies in Java.

Bibliography

[Breshears (2009)] C. Breshears, *The Art of Concurrency*, O'Reilly & Associates (2009).

[Goetz et al. (2006)] B. Goetz, T. Peirls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*, Addison-Wesley (2006).

[Hart (2005)] J. M. Hart, *Windows System Programming*, Third Edition, Addison-Wesley (2005).

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[Oaks (2014)] S. Oaks, *Java Performance—The Definitive Guide*, O'Reilly & Associates (2014).

[Pacheco (2011)] P. S. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann (2011).

[Solomon and Russinovich (2000)] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000,* Third Edition, Microsoft Press (2000).

Chapter 7 Exercises

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

7.8 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

7.9 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

7.10 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 7.14 mainly suitable for small portions.

a. Explain why this is true.

b. Design a new scheme that is suitable for larger portions.

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

7.12 Explain why the call to the `lock()` method in a Java `ReentrantLock` is not placed in the `try` clause for exception handling, yet the call to the `unlock()` method is placed in a `finally` clause.

7.13 Explain the difference between software and hardware transactional memory.

Programming Problems

7.14 Exercise 3.20 required you to design a PID manager that allocated a unique process identifier to each process. Exercise 4.28 required you to modify your solution to Exercise 3.20 by writing a program that created a number of threads that requested and released process identifiers. Using mutex locks, modify your solution to Exercise 4.28 by ensuring that the data structure used to represent the availability of process identifiers is safe from race conditions.

7.15 In Exercise 4.27, you wrote a program to generate the Fibonacci sequence. The program required the parent thread to wait for the child thread to finish its execution before printing out the computed values. If we let the parent thread access the Fibonacci numbers as soon as they were computed by the child thread—rather than waiting for the child thread to terminate—what changes would be necessary to the solution for this exercise? Implement your modified solution.

7.16 The C program `stack-ptr.c` (available in the source-code download) contains an implementation of a stack using a linked list. An example of its use is as follows:

```
StackNode *top = NULL;
push(5, &top);
push(10, &top);
push(15, &top);

int value = pop(&top);
value = pop(&top);
value = pop(&top);
```

This program currently has a race condition and is not appropriate for a concurrent environment. Using Pthreads mutex locks (described in Section 7.3.1), fix the race condition.

7.17 Exercise 4.24 asked you to design a multithreaded program that estimated π using the Monte Carlo technique. In that exercise, you were asked to create a single thread that generated random points, storing the result in a global variable. Once that thread exited, the parent thread performed the calculation that estimated the value of π. Modify that program so that you create several threads, each of which generates random points and determines if the points fall within the circle. Each thread will have to update the global count of all points that fall within the circle. Protect against race conditions on updates to the shared global variable by using mutex locks.

7.18 Exercise 4.25 asked you to design a program using OpenMP that estimated π using the Monte Carlo technique. Examine your solution to that program looking for any possible race conditions. If you identify a race condition, protect against it using the strategy outlined in Section 7.5.2.

7.19 A **barrier** is a tool for synchronizing the activity of a number of threads. When a thread reaches a **barrier point**, it cannot proceed until all other

threads have reached this point as well. When the last thread reaches the barrier point, all threads are released and can resume concurrent execution.

Assume that the barrier is initialized to *N*—the number of threads that must wait at the barrier point:

```
init(N);
```

Each thread then performs some work until it reaches the barrier point:

```
/* do some work for awhile */

barrier_point();

/* do some work for awhile */
```

Using either the POSIX or Java synchronization tools described in this chapter, construct a barrier that implements the following API:

- `int init(int n)`—Initializes the barrier to the specified size.
- `int barrier_point(void)`—Identifies the barrier point. All threads are released from the barrier when the last thread reaches this point.

The return value of each function is used to identify error conditions. Each function will return 0 under normal operation and will return −1 if an error occurs. A testing harness is provided in the source-code download to test your implementation of the barrier.

Programming Projects

Project 1—Designing a Thread Pool

Thread pools were introduced in Section 4.5.1. When thread pools are used, a task is submitted to the pool and executed by a thread from the pool. Work is submitted to the pool using a queue, and an available thread removes work from the queue. If there are no available threads, the work remains queued until one becomes available. If there is no work, threads await notification until a task becomes available.

This project involves creating and managing a thread pool, and it may be completed using either Pthreds and POSIX synchronization or Java. Below we provide the details relevant to each specific technology.

I. POSIX

The POSIX version of this project will involve creating a number of threads using the Pthreads API as well as using POSIX mutex locks and semaphores for synchronization.

The Client

Users of the thread pool will utilize the following API:

- `void pool_init()`—Initializes the thread pool.
- `int pool_submit(void (*somefunction)(void *p), void *p)`—where `somefunction` is a pointer to the function that will be executed by a thread from the pool and `p` is a parameter passed to the function.
- `void pool_shutdown(void)`—Shuts down the thread pool once all tasks have completed.

We provide an example program `client.c` in the source code download that illustrates how to use the thread pool using these functions.

Implementation of the Thread Pool

In the source code download we provide the C source file `threadpool.c` as a partial implementation of the thread pool. You will need to implement the functions that are called by client users, as well as several additional functions that support the internals of the thread pool. Implementation will involve the following activities:

1. The `pool_init()` function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.
2. The `pool_submit()` function is partially implemented and currently places the function to be executed—as well as its data— into a `task` struct. The `task` struct represents work that will be completed by a thread in the pool. `pool_submit()` will add these tasks to the queue by invoking the `enqueue()` function, and worker threads will call `dequeue()` to retrieve work from the queue. The queue may be implemented statically (using arrays) or dynamically (using a linked list).

   The `pool_init()` function has an `int` return value that is used to indicate if the task was successfully submitted to the pool (0 indicates success, 1 indicates failure). If the queue is implemented using arrays, `pool_init()` will return 1 if there is an attempt to submit work and the queue is full. If the queue is implemented as a linked list, `pool_init()` should always return 0 unless a memory allocation error occurs.
3. The `worker()` function is executed by each thread in the pool, where each thread will wait for available work. Once work becomes available, the thread will remove it from the queue and invoke `execute()` to run the specified function.

   A semaphore can be used for notifying a waiting thread when work is submitted to the thread pool. Either named or unnamed semaphores may be used. Refer to Section 7.3.2 for further details on using POSIX semaphores.

4. A mutex lock is necessary to avoid race conditions when accessing or modifying the queue. (Section 7.3.1 provides details on Pthreads mutex locks.)
5. The `pool_shutdown()` function will cancel each worker thread and then wait for each thread to terminate by calling `pthread_join()`. Refer to Section 4.6.3 for details on POSIX thread cancellation. (The semaphore operation `sem_wait()` is a cancellation point that allows a thread waiting on a semaphore to be cancelled.)

Refer to the source-code download for additional details on this project. In particular, the `README` file describes the source and header files, as well as the `Makefile` for building the project.

II. Java

The Java version of this project may be completed using Java synchronization tools as described in Section 7.4. Synchronization may depend on either (a) monitors using `synchronized/wait()/notify()` (Section 7.4.1) or (b) semaphores and reentrant locks (Section 7.4.2 and Section 7.4.3). Java threads are described in Section 4.4.3.

Implementation of the Thread Pool

Your thread pool will implement the following API:

- `ThreadPool()`—Create a default-sized thread pool.
- `ThreadPool(int size)`—Create a thread pool of size `size`.
- `void add(Runnable task)`—Add a task to be performed by a thread in the pool.
- `void shutdown()`—Stop all threads in the pool.

We provide the Java source file `ThreadPool.java` as a partial implementation of the thread pool in the source code download. You will need to implement the methods that are called by client users, as well as several additional methods that support the internals of the thread pool. Implementation will involve the following activities:

1. The constructor will first create a number of idle threads that await work.
2. Work will be submitted to the pool via the `add()` method, which adds a task implementing the `Runnable` interface. The `add()` method will place the `Runnable` task into a queue (you may use an available structure from the Java API such as `java.util.List`).
3. Once a thread in the pool becomes available for work, it will check the queue for any `Runnable` tasks. If there is such a task, the idle thread will remove the task from the queue and invoke its `run()` method. If the queue is empty, the idle thread will wait to be notified when work

becomes available. (The `add()` method may implement notification using either `notify()` or semaphore operations when it places a `Runnable` task into the queue to possibly awaken an idle thread awaiting work.)

4. The `shutdown()` method will stop all threads in the pool by invoking their `interrupt()` method. This, of course, requires that `Runnable` tasks being executed by the thread pool check their interruption status (Section 4.6.3).

Refer to the source-code download for additional details on this project. In particular, the README file describes the Java source files, as well as further details on Java thread interruption.

Project 2—The Sleeping Teaching Assistant

A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

Using POSIX threads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

The Students and the TA

Using Pthreads (Section 4.4.1), begin by creating n students where each student will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.

Perhaps the best option for simulating students programming—as well as the TA providing help to a student—is to have the appropriate threads sleep for a random period of time.

Coverage of POSIX mutex locks and semaphores is provided in Section 7.3. Consult that section for details.

Project 3—The Dining-Philosophers Problem

In Section 7.1.3, we provide an outline of a solution to the dining-philosophers problem using monitors. This project involves implementing a solution to this problem using either POSIX mutex locks and condition variables or Java condition variables. Solutions will be based on the algorithm illustrated in Figure 7.7.

Both implementations will require creating five philosophers, each identified by a number 0 . . 4. Each philosopher will run as a separate thread. Philosophers alternate between thinking and eating. To simulate both activities, have each thread sleep for a random period between one and three seconds.

I. POSIX

Thread creation using Pthreads is covered in Section 4.4.1. When a philosopher wishes to eat, she invokes the function

```
pickup_forks(int philosopher_number)
```

where `philosopher_number` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes

```
return_forks(int philosopher_number)
```

Your implementation will require the use of POSIX condition variables, which are covered in Section 7.3.

II. Java

When a philosopher wishes to eat, she invokes the method `takeForks(philosopherNumber)`, where `philosopherNumber` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes `returnForks(philosopherNumber)`.

Your solution will implement the following interface:

```
public interface DiningServer
{
   /* Called by a philosopher when it wishes to eat */
   public void takeForks(int philosopherNumber);

   /* Called by a philosopher when it is finished eating */
   public void returnForks(int philosopherNumber);
}
```

It will require the use of Java condition variables, which are covered in Section 7.4.4.

Project 4—The Producer–Consumer Problem

In Section 7.1.1, we presented a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 5.9 and 5.10. The solution presented in Section 7.1.1 uses three semaphores: `empty` and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual-exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores for `empty` and `full` and a mutex lock, rather than a binary semaphore, to represent `mutex`. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with the `empty`, `full`, and `mutex` structures. You can solve this problem using either Pthreads or the Windows API.

The Buffer

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

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

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 7.14.

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figure 7.1 and Figure 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the `empty` and `full` semaphores.

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

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

A skeleton for this function appears in Figure 7.15.

```
#include "buffer.h"

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

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

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

Figure 7.14 Outline of buffer operations.

The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand()` function, which produces random integers between 0 and RAND_MAX. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears in Figure 7.16.

```
#include "buffer.h"

int main(int argc, char *argv[]) {
  /* 1. Get command line arguments argv[1],argv[2],argv[3] */
  /* 2. Initialize buffer */
  /* 3. Create producer thread(s) */
  /* 4. Create consumer thread(s) */
  /* 5. Sleep */
  /* 6. Exit */
}
```

Figure 7.15 Outline of skeleton program.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

void *producer(void *param) {
   buffer_item item;

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

void *consumer(void *param) {
   buffer_item item;

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

Figure 7.16 An outline of the producer and consumer threads.

As noted earlier, you can solve this problem using either Pthreads or the Windows API. In the following sections, we supply more information on each of these choices.

Pthreads Thread Creation and Synchronization

Creating threads using the Pthreads API is discussed in Section 4.4.1. Coverage of mutex locks and semaphores using Pthreads is provided in Section 7.3. Refer to those sections for specific instructions on Pthreads thread creation and synchronization.

Windows Threads

Section 4.4.2 discusses thread creation using the Windows API. Refer to that section for specific instructions on creating threads.

Windows Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 7.2.1. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

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

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

In Section 7.2.1, we identified dispatcher objects as being either ***signaled*** or ***nonsignaled***. A signaled dispatcher object (such as a mutex lock) is available for ownership. Once it is acquired, it moves to the nonsignaled state. When it is released, it returns to signaled.

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

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

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

```
ReleaseMutex(Mutex);
```

Windows Semaphores

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

```
#include <windows.h>

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

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

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

```
WaitForSingleObject(Sem, INFINITE);
```

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

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

1. The HANDLE of the semaphore
2. How much to increase the value of the semaphore
3. A pointer to the previous value of the semaphore

We can use the following statement to increase `Sem` by 1:

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

Both `ReleaseSemaphore()` and `ReleaseMutex()` return a nonzero value if successful and 0 otherwise.