CHAPTER 5

Process 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**.

5.1 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 Section 5.1. This section addresses these and other problems while developing a solution using Java synchronization mechanisms.

5.1.1 Busy Waiting and Livelock

Busy waiting was introduced in Section 5.6.2, 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 Section 6.1 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.

```
// 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 5.1 Synchronized `insert()` and `remove()` methods.

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 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.

5.1.2 Race Condition

In Section Section 5.1, we saw an example of the consequences of a race condition on the shared variable `count`. Figure 5.1 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

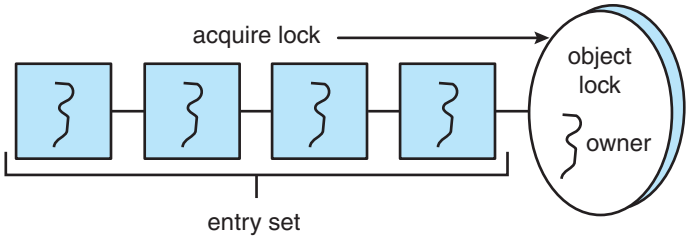

Figure 5.2 Entry set.

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` 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 5.2 illustrates how the entry set operates.

If the producer calls the `insert()` method, as shown in Figure 5.1, 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.

5.1.3 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.

```
// 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 5.3 `insert()` and `remove()` methods using `wait()` and `notify()`.

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.

5.1.4 Wait and Notify

Figure 5.3 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

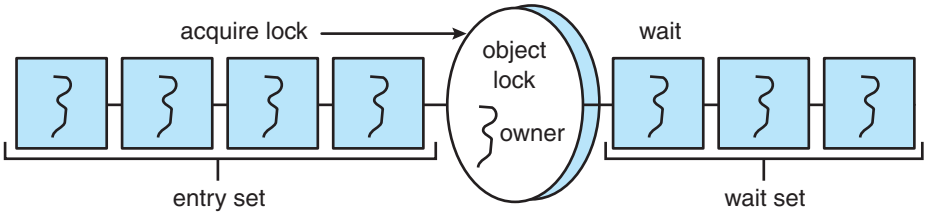

Figure 5.4 Entry and wait sets.

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.

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 5.3. 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 5.4 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 Section 5.6.)

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 5.3. 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 5.5 represents the complete solution to the bounded-buffer problem using Java synchronization. This class

```
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 Figure 5.3
    }

    public synchronized E remove() {
       // Figure Figure 5.3
    }
}
```

Figure 5.5 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 5.6 `doWork()` method.

may be substituted for the `BoundedBuffer` class used in the semaphore-based solution to this problem in Section Section 5.7.1.

5.2 Multiple Notifications

As described in Section Section 5.1.4, 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 5.6. 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

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

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

    public synchronized void acquireReadLock() {
       // Figure Figure 5.8
    }

    public synchronized void releaseReadLock() {
       // Figure Figure 5.8
    }

    public synchronized void acquireWriteLock() {
       // Figure Figure 5.9
    }

    public synchronized void releaseWriteLock() {
       // Figure Figure 5.9
    }
}
```

Figure 5.7 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 5.8 Methods called by readers.

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.

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()`.

5.3 A 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 5.7. 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

```
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 5.9 Methods called by writers.

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 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.

5.4 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 5.10. Access to the `criticalSection()` method in Figure 5.10 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 5.11.

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

    synchronized(mutexLock) {
      criticalSection();
    }

    remainderSection();
}
```

Figure 5.10 Block synchronization.

5.5 Synchronization Rules

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

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.

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

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

Figure 5.11 Block synchronization using `wait()` and `notify()`.

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 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
    */
}
```

5.6 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.

5.7 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.

5.7.1 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 Section 5.1.2: 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 Section 5.1.2 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.

5.7.2 Semaphores

The Java 5 API also provides a counting semaphore, as described in Section Section 5.6. 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 (Section Section 5.6). 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.

5.7.3 Condition Variables

The last utility we cover in the Java API is the condition variable. Just as the ReentrantLock (Section Section 5.7.1) 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 Section 5.8.

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 pro-

vide 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 Section 5.8, 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 5.6 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 5.6 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 5.12. 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()`.

Exercises

5.1 The **Singleton** design pattern ensures that only one instance of an object is created. For example, assume we have a class called `Singleton` and we wish to allow only one instance of it. Rather than creating a `Singleton` object using its constructor, we instead declare the constructor as `private` and provide a public static method—such as `getInstance()` —for object creation:

```
Singleton sole = Singleton.getInstance();
```

Figure 5.13 provides one strategy for implementing the Singleton pattern. The idea behind this approach is to use **lazy initialization**,

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

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

      // 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;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```

Figure 5.12 `doWork()` method with condition variables.

whereby we create an instance of the object only when it is needed—that is, when `getInstance()` is first called. However, Figure 5.13 suffers from a race condition. Identify the race condition.

Figure 5.14 shows an alternative strategy that addresses the race condition by using the **double-checked locking idiom**. Using this strategy, we first check whether `instance` is `null`. If it is, we next obtain the lock for the `Singleton` class and then double-check whether `instance` is still `null` before creating the object. Does this strategy result in any race conditions? If so, identify and fix them. Otherwise, illustrate why this code example is thread-safe.

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton { }

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();

      return instance;
   }
}
```

Figure 5.13 First attempt at Singleton design pattern.

Programming Problems

5.2 Exercise 4.6 requires the main thread to wait for the sorting and merge threads by using the `join()` method. Modify your solution to this exercise so that it uses semaphores rather than the `join()` method. (Hint: We recommend carefully reading through the Java API on the constructor for `Semaphore` objects.)

5.3 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections open at any point in time. After N connections have been made, the server will not accept another incoming connection until an existing connection is released. In the source code download, there is a program named `TimedServer.java` that listens to port 2500. When a connection is made (via telnet or the supplied client program `TimedClient.java`),

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton { }

   public static Singleton getInstance() {
      if (instance == null) {
         synchronized(Singleton.class) {
            if (instance == null)
               instance = new Singleton();
         }
      }

      return instance;
   }
}
```

Figure 5.14 Singleton design pattern using double-checked locking.

the server creates a new thread that maintains the connection for 10 seconds (writing the number of seconds remaining while the connection remains open). At the end of 10 seconds, the thread closes the connection. Currently, `TimedServer.java` will accept an unlimited number of connections. Using semaphores, modify this program so that it limits the number of concurrent connections.

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

The following Java class is used to manage a finite number of instances of an available resource. Note that when a process wishes to obtain a number of resources, it invokes the `decreaseCount()` method. Similarly, when a process wants to return a number of resources, it calls `increaseCount()`.

```
public class Manager
{
   public static final int MAX_RESOURCES = 5;
   private int availableResources = MAX_RESOURCES;

   /**
    * Decrease availableResources by count resources.
    * return 0 if sufficient resources available,
    * otherwise return -1
    */
   public int decreaseCount(int count) {
      if (availableResources < count)
         return -1;
      else {
         availableResources -= count;

         return 0;
      }

   /* Increase availableResources by count resources. */
   public void increaseCount(int count) {
      availableResources += count;
   }
}
```

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

a. Identify the data involved in the race condition.

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

c. Using Java synchronization, fix the race condition. Also modify `decreaseCount()` so that a thread blocks if there aren't sufficient resources available.

5.5 Implement the `Channel` interface (Figure 3.7) so that the `send()` and `receive()` methods are blocking. That is, a thread invoking `send()` will block if the channel is full. If the channel is empty, a thread invoking `receive()` will block. Doing this will require storing the messages in a fixed-length array. Ensure that your implementation is thread-safe (using Java synchronization) and that the messages are stored in FIFO order.

5.6 A **barrier** is a thread-synchronization mechanism that allows several threads to run for a period and then forces all threads to wait until all have reached a certain point. Once all threads have reached this point (the barrier), they may all continue. An interface for a barrier appears as follows:

```
public interface Barrier
{
   /**
    * Each thread calls this method when it reaches
    * the barrier. All threads are released to continue
    * processing when thelast thread calls this method.
    */
   public void waitForOthers();

   /**
    * Release all threads from waiting for the barrier.
    * Any future calls to waitForOthers() will not wait
    * until the Barrier is set again with a call
    * to the constructor.
    */
   public void freeAll();
}
```

The following code segment establishes a barrier and creates 10 `Worker` threads that will synchronize according to the barrier:

```
public static final int THREAD_COUNT = 10;

Barrier jersey = new BarrierImpl(THREAD_COUNT);
for (int i = 0; i < THREAD_COUNT; i++)
   (new Worker(jersey)).start();
```

Note that the barrier must be initialized to the number of threads that are being synchronized and that each thread has a reference to the same barrier object—`jersey`. Each `Worker` will run as follows:

```
// All threads have access to this barrier
Barrier jersey;

// do some work for a while . . .

// now wait for the others
jersey.waitForOthers();

// now do more work . . .
```

When a thread invokes the method `waitForOthers()`, it will block until all threads have reached this method (the barrier). Once all threads have reached the method, they may all proceed with the remainder of their code. The `freeAll()` method bypasses the need to wait for threads to reach the barrier; as soon as `freeAll()` is invoked, all threads waiting for the barrier are released.

Implement the `Barrier` interface using Java synchronization.

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

Programming Projects

The projects below deal with three distinct topics—designing a pid manager, designing a thread pool, and implementing a solution to the dining-philosophers problem using Java's condition variables.

Project 1: Designing a pid Manager

A pid manager is responsible for managing process identifiers (pids). When a process is first created, it is assigned a unique pid by the pid manager. The pid is returned to the pid manager when the process completes execution. The pid manager may later reassign this pid. Process identifiers are discussed more fully in Section Section 3.3.1. What is most important here is to recognize that process identifiers must be unique; no two active processes can have the same pid.

The Java interface shown in Figure 5.15 identifies the basic methods for obtaining and releasing a pid. Process identifiers are assigned within the

```
/**
 * An interface for a PID manager.
 *
 * The range of allowable PID's is
 * MIN_PID .. MAX_PID (inclusive)
 *
 * An implementation of this interface
 * must ensure thread safety.
 */

public interface PIDManager
{
   /** The range of allowable PIDs (inclusive) */
   public static final int MIN_PID = 4;
   public static final int MAX_PID = 127;

   /**
    * Return a valid PID or -1 if
    * none are available
    */
   public int getPID();

   /**
    * Return a valid PID, possibly blocking the
    * calling process until one is available.
    */
   public int getPIDWait();

   /**
    * Release the pid
    * Throw an IllegalArgumentException if the pid
    * is outside of the range of PID values.
    */
   public void releasePID(int pid);
}
```

Figure 5.15 Java interface for obtaining and releasing a pid.

range `MIN_PID` to `MAX_PID` (inclusive). The fundamental difference between `getPID()` and `getPIDWait()` is that if no pids are available, `getPID()` returns `-1`, whereas `getPIDWait()` blocks the calling process until a pid becomes available. As with most kernel data, the data structure for maintaining a set of pids must be free from race conditions and deadlock. One possible result from a race condition is that the same pid will be concurrently assigned to more than one process. (However, a pid can be reused once it has been returned via the call to `releasePID()`.) To achieve blocking behavior in `getPIDWait()`, you may use any of the Java-based synchronization mechanisms discussed in this chapter.

Project 2: Designing a Thread Pool

Create a thread pool (see Chapter 4) using Java synchronization. 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 stopPool()`	Stop all threads in the pool

Your pool will first create a number of idle threads that await work. 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. Once a thread in the pool becomes available for work, it will check the queue for any `Runnable` tasks. If there are such tasks, 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 will perform a `notify()` when it places a `Runnable` task into the queue to possibly awaken an idle thread awaiting work.) The `stopPool()` method will stop all threads in the pool by invoking their `interrupt()` method (Section Section 4.5). This, of course, requires that `Runnable` tasks being executed by the thread pool check their interruption status.

There are many different ways to test your solution to this problem. One suggestion is to modify your answer to Exercise Exercise 3.3 so that the server can respond to each client request by using a thread pool.

Project 3: Dining Philosophers

In Section Section 5.8.2, we provide an outline of a solution to the dining-philosophers problem using monitors. This exercise will require implementing this solution using Java's condition variables.

Begin by creating five philosophers, each identified by a number 0...4. Each philosopher runs as a separate thread. Philosophers alternate between thinking and eating. When a philosopher wishes to eat, it invokes the method `takeForks(philNumber)`, where `philNumber` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, it invokes `returnForks(philNumber)`.

Your solution will implement the following interface:

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

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

The implementation of the interface follows the outline of the solution provided in Figure 5.18. Use Java's condition variables to synchronize the activity of the philosophers and prevent deadlock.