

Chapter 3



Race Conditions and Mutual Exclusion

- ▼ KNOW YOUR ENEMY!
- ▼ RACE CONDITIONS
- ▼ CONDITIONS
- ▼ CLASSIC SYNCHRONIZATION MECHANISMS
- ▼ EXERCISES

Java is, without a doubt, popular because of its extensive support for multithreading and networking. It is also among an elite group of languages that directly support concurrent and distributed computing right out of the box. Despite this support, Java is plagued by the same problems that have traditionally affected programmers who write concurrent programs. This chapter is dedicated to *race conditions*, which undoubtedly account for most of the problems that occur in concurrent programming.

We will begin with a discussion of the general “enemies” that manifest themselves in concurrent programming, and follow it with a discussion of a specific enemy—the race condition. We then turn to a discussion of how to eliminate race conditions by using mutual exclusion and built-in object synchronization (which are supported by each and every Java object, regardless of whether it is used or not). The chapter concludes with a discussion of the need for more complex synchronization control with conditions and a brief tour of some classic high-level synchronization mechanisms found in computer science literature on operating systems, programming languages, and parallel processing.



Know Your Enemy!

Concurrency creates problems unlike those you encounter with single processes. You must know how to counteract these problems in order to use threads successfully. There are three problems in particular to worry about:

- *Race conditions*: Threads can try to update the same data structure at the same time. The result can be partly what one thread wrote and partly what the other thread wrote. This garbles the data structure, typically causing the next thread that tries to use it to crash.
- *Deadlock*: To avoid updating the same data structure at the same time, threads lock a data structure until they have finished making their modifications to it. Other threads are forced to wait to modify a data structure until it is unlocked. However, if threads try to modify two data structures at the same time, they will also try to lock them both. Thread *A* can lock data structure *X*, and thread *B* can lock data structure *Y*. Then, if *A* tries to lock *Y* and *B* tries to lock *X*, both *A* and *B* will wait forever—*B* will wait for *A* to unlock *X* and *A* will wait for *B* to unlock *Y*. This deadlock occurs easily, and you must find ways to prevent it.
- *Starvation*: In its effects, starvation is similar to deadlock in that some threads do not make progress. But the causes of starvation are different from those of deadlock: The threads could, in theory, execute, but they just don't get time on a processor, perhaps because they don't have a high enough priority.
- Other forms of *nondeterminism*: Race conditions are the most common form of nondeterminism encountered in practical multithreading problems; however, other forms of nondeterminism also occur in practice. For example, two different executions of a multithreaded program may run correctly, but produce output or traces that appear to be different from one another (i.e., sometimes called different serializations). This is a case that occurs in transaction systems.

Race Conditions

Egoist: A First Study in Race Conditions

To motivate the problem of race conditions, consider the example shown in Example 3-1, which presents two classes, `Egoist` and `OutChar`. An `egoist` is an object that has a name and contains a reference to an `OutChar` instance. When executed, the `run()` method (the body of the thread) will simply print the name of the `egoist` repeatedly (300 times) to the output stream.

The `OutChar` class is a wrapper class that is used to print characters one at a time and limit the output to 50 characters per line. The `out()` method is marked “synchronized,” because the `OutChar` instance will be shared by multiple threads.



Example 3-1 presents a class, `Sched`, that makes use of both `Egoist` and `OutChar`. Three `Egoist` instances (A, B, and C) are created in the `main()` method. A single `OutChar` instance is also created, which is referenced by the three `Egoist` instances. Once the `Egoist` has been instantiated, its priority level is set. (See “A Word on Priorities” on page 46.)

This example shows the effect of thread scheduling on the outcome of a given program’s execution. Recall that, in the introductory discussion about “knowing your enemy,” we discussed how concurrent programming with threads introduces the possibility of “other forms of nondeterminism.” In this case, the output can be in any order, because the `OutChar` object (variable C in the code) only guarantees synchronized access to its state when a single character is output. It is possible to serialize the accesses of the individual `Egoist` objects to the single `OutChar` object; however, that has not been done in this code.

However, there is a question, that arises as a result of this example:

Why isn’t the output always the same?

The answer is not a simple one and raises several additional questions:

- **Just exactly what *does* happen when the code `sleep(5)` is executed?** This code is used in the thread class to put the currently running thread to sleep for 5 milliseconds. Is it possible that the thread “sleeps” for more than 5 milliseconds? The answer is yes and ultimately depends on the operating system being used. Time-sharing operating systems (UNIX and WindowsNT) make no guarantees about how long a thread (or a process) will be away from the CPU once it yields to the CPU. The sleep call results in the current thread blocking immediately, thus yielding to the CPU.
- **What role, if any, is played by the `setPriority` calls?** Again, the answer depends on the operating system. Thread scheduling priorities are a highly OS-dependent phenomenon. Windows NT thread scheduling is limited to five priority levels (at least in Version 4.0 and earlier releases). Some versions of UNIX don’t support threads, at all. Solaris and Linux, which both support the `pthread`s standard, support at least 255 priority levels. Most thread implementations do support the notion of a highest-priority thread, which, if running, is allowed to run until it yields to the CPU.
- **Is thread scheduling preemptive?** This is an important question. Most operating systems that support threads do support preemptive thread scheduling.
- **Is there kernel-level or user-level scheduling?** It turns out that this is an important question to answer in a general sense. The answers most likely will not affect your view of the outcome. However, if you run this code on a machine with multiple CPUs, the outcome could be very different, because it is possible that more than one thread



can be running on two different CPUs. Most Java implementations do not yet support the notion of symmetric scheduling; but by the time you are reading this, it is more than likely that Java implementations will exist that support parallel hardware well.

Example 3-1 sched - The Driver for Egoist to Demonstrate Scheduling

```
class Sched extends Thread {
    public void run(){
        for(;;) {
            try {
                sleep(5);
            }
            catch(InterruptedException e) { }
        }
    }

    public static void main(String[] s) {
        Thread q; int i;
        OutChar C=new OutChar();
        for (i='A';i<='C';i++) {
            q=new Egoist((char) i,C);
            q.setPriority(Thread.NORM_PRIORITY-1);
            // try commenting the above out
            q.start();
        }
        q=new Sched();
        q.setPriority(Thread.MAX_PRIORITY);
        q.setDaemon(true);
        q.start();
    }
}
```

A Word on Priorities

Before continuing with the discussion of race conditions and presenting more examples, a few words are warranted about Java thread priority levels. The `Thread` class provides the definitions of two constants: `MIN_PRIORITY` and `MAX_PRIORITY`. If you check the Java Development Kit source code, the values of these constants are set to 1 and 10, respectively. The choice of these particular numbers is somewhat curious, since the use of thread priority levels is somewhat common in actual programming practice.

In parallel algorithms (e.g., branch-and-bound problems or game trees, as in chess), there may be situations in which hundreds of possible alternatives can be evaluated concur-



rently. Some of the alternatives are likely to require further exploration, while others clearly do not. Suppose you have the resources (i.e., dozens of processors) to evaluate many alternatives. You would clearly want the ability to evaluate the alternatives that are most likely to yield a good result. (In chess, your hope as a programmer is to choose alternatives that lead to victory.) Based on an evaluation function, you could assign a priority value to the thread that will explore the alternatives further. Priorities allow the alternatives to be scheduled dynamically, working from the best to the worst. As more alternatives are discovered, it is possible that more threads will be created, again with dynamically assigned priorities. As desirable alternatives are discovered, they will be scheduled ahead of undesirable alternatives.

Now, let us return to the point: Thread priorities *are* useful! Since they are known to be useful, it would be convenient if Java supported more than 10 priority levels, though there does seem to be a reason for the low number. As mentioned earlier, Windows NT supports five levels, and `pthread`s (the UNIX threads standard) supports many. There is no clear explanation provided in the Java specification; however, the choice of 10 seems particularly close to Windows NT. This fact, in and of itself, might provide some insight as to why there are so few, since Java has been designed for platform independence. Platform independence sometimes comes at the cost of utilizing the full power of a particular platform. Later in this book, we will return to the issue of thread priorities and examine how one can support an arbitrary number of priority levels for certain kinds of tasks, similar to Example 3-1.

What is a Race Condition?

The `Egoist` example showed the results of a race condition in which the output depended on the scheduling of the execution of the concurrent threads. A *race condition* typically is a result of nondeterministic ordering of a set of external events. In the case of `Egoist`, the events were the ticks of a computer clock that caused rescheduling. The ticks of a computer clock are used in most operating systems (like UNIX and Windows NT) to support preemptive task scheduling. A time slice (discussed earlier) is usually computed as a function of so many clock ticks. In dealing with networking applications, the external events include the speed and scheduling of both the remote computers you are linked to and queueing delays on the network.

In a concurrent application, race conditions are not always a sign of problems. There may be many legitimate execution orders (often referred to in the computing literature as *serializations* or *schedules*) for the same program, all of which produce an equivalent and correct result. Consider the concept of sorting an array or list of values, to be rearranged such that the elements are in either ascending or descending order. One way to perform a concurrent sort is to break up the array into small chunks, sort each chunk using a well-known sequential algorithm (like selection sort), and then merge all of the small arrays, in pairwise fashion, until there are no more merges to be performed. There are many possible



schedules for performing the merge, all of which will lead to the correct result (provided the merging algorithm is correct). This conceptual example, which actually appears as a programming example later in the book, leads to an important observation about concurrent programming in general: Proving the correctness of a concurrent program is harder than doing so for one that is not concurrent, or sequential. For a concurrent program to be correct, all serializations of the program must yield a correct result. The notion of a correct result is something we will be addressing throughout this book. Clearly, if any serialization leads to the values of the array not being sorted for some reason, the program is not correct. Another example of an incorrect result is if the program does not terminate. This could occur in the presence of deadlock, the subject of a later chapter.

The most dangerous race conditions, however, involve access to shared data structures. If two threads are updating the same data structure at the same time, the changes may be made partially by one thread and partially by the other. The contents of the data structure can then become garbled, which will confuse threads that access it later, thereby causing them to crash. The trick is to force the threads to access the data structure one at a time, so-called mutual exclusion, so that each thread can complete its update and leave the structure in a consistent state for the next thread.

To do this, threads must be able to lock the data structure while it is being accessed and then unlock it when they are finished. Code that updates a shared data structure is called a *critical section*.

Consider again the `Egoist` example. This code showed an example of a harmless race condition. Each `Egoist` instance outputs a single character (its name) to the output stream a total of 300 times. One expects to see the letters “A,” “B,” and “C” each appear 300 times. If, for some reason, each of the letters did not appear 300 times, it could be concluded that there is, indeed, a bug in the code. The way the sample code has been written, there is no implicit or explicit design that will make the output appear in any particular order (or be grouped in any particular way). The only thing that is assured is that each `Egoist`’s name appears 300 times and that no fewer (or greater) than 50 characters per line is output.

Race0 Class

Let us now consider an example in which race conditions can cause something unexpected to occur. Example 3-2 demonstrates an example of two threads that are created to both be able to access a common data structure. The data structure to be shared between the two threads is aptly named `Shared0`, shown in Example 3-3. As the name indicates, an instance of this class will be somehow shared. `Shared0` consists of two instance variables, `x` and `y`, both of type integer, and two methods:


Example 3-2 A Simple Example of Race Conditions (Race0.java)

```

class Race0 extends Thread {
    static Shared0 s;
    static volatile boolean done=false;

    public static void main(String[]x) {
        Thread lo=new Race0();
        s=new Shared0();
        try {
            lo.start();
            while (!done) {
                s.bump();
                sleep(30);
            }
            lo.join();
        } catch (InterruptedException e)
            { return; }
    }

    public void run() {
        int i;
        try {
            for (i=0;i<1000;i++) {
                if (i%60==0)
                    System.out.println();
                System.out.print(".X".charAt(s.dif()));
                sleep(20);
            }
            System.out.println();
            done=true;
        } catch (InterruptedException e)
            { return; }
    }
}

```

- **bump()**: When called, this method increments the value of *x*, puts the currently running thread to sleep for awhile, and then increments the value of *y*. The outcome of the `bump()` method is that both variables are incremented by one.
- **dif()**: When called, this method returns the difference between the current values of *x* and *y*.

The class responsible for accessing the instance of `Shared0` is `Race0`. `Race0` has no instance variables but does declare two static variables. Static variables are variables that are shared (and visible) among all instances of the class `Race0`, as well as the `main()`



Example 3-3 A Class That will be Used to Create a Shared Data Objects.

```
class Shared0 {
    protected int x=0,y=0;

    public int dif() {
        return x-y;
    }

    public void bump() throws InterruptedException {
        x++;
        Thread.sleep(9);
        y++;
    }
}
```

method defined in `Race0`. (That is, an instance is not even required to access the variable `s`, as the `main()` method illustrates.) There are two methods here:

- `main()`: The method is the main thread of execution. It creates the shared object, starts a second thread, and immediately enters a loop that executes until a variable `done` is set true (by the other thread). Inside the loop, the `bump()` method is repeatedly called on by the shared object `s`, and then the thread goes to sleep for awhile. Once the loop is done, the `join()` method is called on the other thread to await its completion. `join()` is a very convenient way to wait for a thread to finish its execution, but it must be used with care. We give more details on this later.
- `run()`: the routine that is called when a thread is scheduled for execution. Recall that the `main()` method called `lo.start()` to start the other thread. The other thread is an instance of `Race0`. Thus, `Race0` must have a `run` method to ensure that some task is actually performed. This method enters a loop that, similar to the `Egoist` program, executes a certain number of times and formats the output such that only 60 characters per line are written. The `dif()` method is called to determine the difference between `x` and `y`, as explained earlier, and prints “.” or “X.” Then this thread sleeps for awhile. When the loop is finished, the `done` variable is set so that the main thread can terminate its `while` loop.

The box titled “A First Run of `Race0`” illustrates the race condition clearly. Sometimes the difference is 0 and other times 1. The difference is never a value other than 0 or 1. Why is this the case?



A First Run of Race0

```
java Race0

.....X.X.X.....X.X.X.....X.....X.X.X.X.X.X.....
.....X.X.X.X.X.....X.X.X.X.X.....
X.X.X.X.....X.X.X.X.X.....X.X.X.X.....
.....X.X.X.....X.X.X.X.X.....X.X.X.X.X.....
.....X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.....
.X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.....
.....X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.....
.X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.....
X.X.....X.X.X.X.X.....X.X.X.X.....X.X.X.X.....X
X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.....X.X.X.X.....
.....X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.....
.....X.X.X.X.....X.X.X.X.....X.X.X.X.....X.....
X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.....X.....
.....X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.....
X.X.X.X.X.....X.X.X.X.X.....X.X.X.X.X.X
```

- A key to understanding concurrency is to fully understand what would happen if all traces of concurrency were removed. For example, what would happen if we called `bump()` and `dif()`, guaranteeing in the process that the two could never execute at the same time? The difference between `x` and `y` would always be 0. The fact that it is sometimes 1 implies that `dif()` must somehow be able to execute in the middle of the execution of the `bump()` method.
- The `bump()` method always increments `x` and `y`. The value of `y` can never be different from that of `x` by more than 1.
- The `dif()` method never updates `x` or `y`. `bump()` is the only method that does so.
- Furthermore, in this example, only one call of `bump()` or `dif()` is active at any given time; however, due to multithreading, both `bump()` and `dif()` may sometimes be running simultaneously.

The bottom line is that in order to fully understand a concurrent program, even one as simple as this, you must consider the serial behavior and understand exactly what could be happening. In the preceding code, it is possible for the value of `x` or `y` to be read by the thread calling `dif()` in the middle of the `bump()` call. This is why the difference is sometimes 1 and at other times 0.



The box titled “A Second Run of Race0” shows a second run of the Race0 code. This example demonstrates that attempting to predict the pattern is somewhat of an exercise in futility.

A Second Run of Race0

```
java Race0

.....X.X.X.....X.X.X.....X.X.X.X.X.X.X.....X.X.
X.X.X.X.....X.X.X.X.....X.X.X.....X.X.X.....
.....X.X.X.X.X.X.....X.X.X.X.....X.X.X.X
.X.X.....X.X.X.X.....X.X.X.X.....
.X.X.X.X.X.X.....X.X.X.X.X.X.....X.X.
X.X.X.X.....X.X.X.X.X.X.....X.X.X.X.X.
X.....X.X.X.X.X.X.....X.X.X.X.X.X.....
.....X.X.X.X.X.X.....X.X.X.X.X.X.....
.....X.X.X.X.X.X.....X.X.X.X.X.X.....X
.X.X.X.X.X.....X.X.X.X.X.X.....X.X.X.X
.....X.X.X.X.X.X.....X.X.X.X.X.X.....
.....X.X.X.X.X.X.....X.X.X.X.X.X.....
.....X.X.X.X.X.X.....X.X.X.X.X.X.....
X.....X.X.X.X.X.X.....X.X.X.X.X.X.....
.....X.X.X.X.X.X.....X.X.X.X.X.X.....
.....X.X.X.X.X.X.....X.X.X.X.X.....
```

A cursory look at the output might lead you to think that the results of the two runs are the same. The output patterns are similar but not the same. This is clearly an artifact of scheduling. The long stretches of dots (“.”) indicate that often, scheduling strictly alternates between the thread causing the bump and the thread computing the difference. At other times there is a pattern of strict alternation between equal and unequal values of *x* and *y*. To fully understand the patterns, you would need to consider how the operating system schedules threads and how the sleep time is computed. The fact that there are patterns is not entirely coincidental, but is not predictable either.

Critical Sections and Object Locking

What we have shown in Example 3-2 and 3-3 demonstrates a concept known as a *critical section*. A critical section is a section of code that accesses a common data structure. Shared0 was the class used to create an object that is shared between the two running threads. Because Java is object oriented, the data structures are usually encapsulated in individual objects. It is not always a requirement that an object be used. Scalar values (e.g., int, float, and char variables) may also be used without being “wrapped” by using a class.



The term “critical section” has its origins in the operating systems community. As mentioned in Chapter 1, problems of concurrency go all the way back to programming with multiple processes. (UNIX, an operating system with a design that goes back to the late 1960s and early 1970s, has provided full support for concurrent programming from its inception.) In many respects, the term “section” is a bit of a misnomer. As is often the case in actual practice, multiple “sections,” or blocks, of code access common variables or shared data structures. As you are reading, bear in mind that a critical section will usually be guarded by a common synchronization mechanism.

Java handles critical sections by allowing threads to lock individual objects. In Java, every object has a lock which can be accessed by any piece of code that merely holds a reference to an object. To lock an object, Java has synchronized statements and declarations, which are easily spotted in code with the keyword `synchronized`. If you use the synchronized statement, the object is locked for the duration of your execution of the statement.

It is very common to use synchronization (in particular the Java programming library itself) in method declarations. Methods that are declared to be synchronized will result in the entire object being locked for the duration of the call. There are some subtleties to understanding what “the duration of a call” means, primarily due to exceptions. An exception may cause transfer of control to leave a function in much the same way as an explicit return statement or reaching the end of a `void` method call does. All forms of exiting a procedure call result in proper release of the lock, which is very convenient for programmers. In other languages, such as C and C++, which require the use of an external threads library that is not part of the language proper, a great deal of care was once required to ensure that a lock was explicitly released *prior* to leaving the function.

Race1 Class—Fixing Race0 with Synchronization

Example 3-4 shows a reworked version of class `Race0`, which has been renamed `Race1`. Aside from making use of class `Shared1`, `Race1` remains fundamentally unchanged from `Race0`. The major changes have taken place in class `Shared1`, which is the reworked version of the class `Shared0`, found in Example 3-5. The key difference between `Race0` and `Race1` is that the methods `bump()` and `dif()` are now synchronized methods. This means that the shared object is now locked whenever `bump()` or `dif()` is executed. Again, this means that `bump()` and `dif()` will not be able to “use” the shared object simultaneously, since they are each declared synchronized. If either the `bump()` or `dif()` method is called while the other is executing, the called method cannot proceed until the other one has completed its work.

Note that both `bump()` and `dif()` must be declared synchronized. If a method or statement is not declared synchronized, it won’t even look at the lock before using the object. Note also that there is no convenient shorthand to indicate that all of the methods in a class are to be synchronized (i.e., there is no such thing as a synchronized “class”).



Example 3-4 Eliminating Race Condition in Race0 (Race1.java)

```
class Race1 extends Thread {
    static Shared1 s;
    static volatile boolean done=false;

    public static void main(String[] args) {
        Thread lo=new Race1();
        s=new Shared1();
        try {
            lo.start();
            while (!done) {
                s.bump();
                sleep(30);
            }
            lo.join();
        } catch (InterruptedException e) {return;}
    }

    public void run() {
        int i;
        try {
            for (i=0;i<1000;i++) {
                if (i%60==0) System.out.println();
                System.out.print(".X".charAt(s.dif()));
                sleep(20);
            }
            System.out.println();
            done=true;
        } catch (InterruptedException e) {return;}
    }
}
```

Example 3-5 Eliminating Race Condition in Shared1 (Race1.java)

```
class Shared1 {
    protected int x=0,y=0;
    public synchronized int dif() {
        return x-y;
    }
    public synchronized void bump() throws
        InterruptedException {
        x++;
        Thread.sleep(9);
        y++;
    }
}
```



Example 3-6 FileCopy0: A Single-Threaded File Copying Program

```
import java.util.*;
import java.io.*;

public class FileCopy0 {

    public static final int BLOCK_SIZE = 4096;

    public static void copy(String src, String dst)
        throws IOException {
        FileReader fr = new FileReader(src);
        FileWriter fw = new FileWriter(dst);
        char[] buffer = new char[BLOCK_SIZE];
        int bytesRead;

        while (true) {
            bytesRead = fr.read(buffer);
            System.out.println(bytesRead + " bytes read");
            if (bytesRead < 0)
                break;
            fw.write(buffer, 0, bytesRead);
            System.out.println(bytesRead + " bytes written");
        }
        fw.close();
        fr.close();
    }

    public static void main(String args[]) {
        String srcFile = args[0];
        String dstFile = args[1];
        try {
            copy(srcFile, dstFile);
        } catch (Exception e) {
            System.out.println("Copy failed.");
        }
    }
}
```

appear in the shared data structure, then writes bytes to the destination (copy) file. The producer and consumer are to be concurrently started as threads.

This type of application is difficult to write by simply relying on the basic synchronization features (locking) found in Java. Recall that both the producer and consumer are started as threads. This means that either thread has a chance of being scheduled first (in general), which results in a race to access the shared data structure. From the onset, there is the pos-



sibility that the consumer gets there first. The consumer locks the shared data structure and tests whether there is a block to be written. If there is, the block is removed from the shared data structure and appended to the destination file. If there is no block to be written, the data structure is left alone. In either case, the shared data structure is unlocked, giving the producer a chance to write a block. The producer must go through the similar lock and unlock maneuver. When the producer acquires the lock, it can write a block into the shared data structure, assuming that there is room to write the block. (There will be room if the data structure is dynamic as opposed to fixed.) Assuming that the producer was able to write the block, the consumer will eventually be able to read a block from the shared data structure.

This example shows the unwieldy nature of working exclusively with synchronization. Synchronization itself allows you to protect shared data, but does very little for elegantly passing control between the producer and consumer. This is because the consumer needs to be able to test whether a block has, in fact, been written. Instead of going through the endless cycle of acquiring the lock, testing the condition, and then releasing the lock, it would be nice if the consumer could temporarily give up the lock and wait until (hopefully) another thread runs and causes the state of the shared data structure to change in a favorable way to the consumer.

Key Object Methods Needed to Work with Conditions in Java

Java provides built-in support awaiting this “change in state” via the notion of a condition. A condition is a bit of a misnomer, however, because it is entirely up to the user whether or not a condition actually occurred. Furthermore, a condition need not be specifically true or false.

To use conditions, one must become familiar with three key methods of the `Object` class:

- `wait()`: This method is used to await a condition. It is called when a lock is presently being held for a particular (shared) object.
- `notify()`: This method is used to notify a single thread that a condition has (possibly) changed. Again, this method is called when a lock is presently being held for a particular object. Only a single thread can be awakened as a result of this call.
- `notifyAll()`: This method is used to notify multiple threads that a condition has (possibly) changed. All threads that are running at the time this method is called will be notified.

`notify()` and `notifyAll()` are *memoryless* operations. If no threads are waiting, then none are awakened. Java does not remember a `notify()` and therefore cannot use it to awaken a thread that waits later.



These three methods are available to any class, since all classes, in one way or another, are derived from the standard Java `Object` class.

Although a thread may be waiting for a particular condition to become true, there is no way to specify the condition in the `wait()` or in the `notify()`. Even if there were only one condition a thread could wait for in a particular object, it is not guaranteed that the condition will be true when the thread starts executing again: After it has been notified, a thread still may not start executing immediately. Another thread may lock the object and make the condition false before the notified thread can run.

This is sometimes called the “false wakeup problem.” It can be a potentially confusing problem on an SMP system, since on these systems, there is a real possibility that two or more threads are awakened simultaneously.

On a condition variable, the proper way to wait is to wait inside a loop. This is done as follows:

```
while (!condition) wait();
```

At first, this technique appears to be awkward; however, it is the safest way to work with threads. Most of the time, the loop only executes for one iteration. When the loop is exited, it is without doubt that the condition being tested is true.

Also, `wait()` can be terminated by an `interrupt` from another thread, so you will have to put the command in a `try` statement.

File Copying: A Producer–Consumer Example

Until now, we have considered only many small examples of important conceptual value, but somewhat limited practical value. Here, we consider a more practical example of how threads, synchronization mechanisms, and conditions all come together in order to support a practical need: file copying.

While this example has been structured to be useful for a reference textbook, it demonstrates a number of useful building blocks that could be adapted for more sophisticated applications. In the approach to file copying presented here, concurrency is used with the hope of improving file copying performance, especially in an environment in which resources may enable this possibility. (e.g., multiprocessor systems, multiple I/O devices, multiple I/O buses, etc.)

If you want to take advantage of concurrency, more often than not it pays to have multiple processors; however, in the case of file copying, you may even benefit from concurrency on a single-processor architecture. Traditionally, file copying is done either a byte at a time (a naive implementation) or a block at a time (somewhat more sophisticated). Copying one byte at a time is not efficient, because block devices themselves communicate (via device drivers) with the operating system one block at a time. (Common block sizes are 512 and



1,024 bytes.) However, even when file copying is done a block at a time, there are some important aspects to consider:

- Reads are typically faster than writes. This tends to be true of most devices and is more obvious in the case of floppy disks and recordable compact discs.
- In both reading and writing, the process of finding the block tends to cause a zigzag pattern of accesses.

In short, the reader and writer are usually not working at the same rate. When you approach this problem serially, the process of alternating between reading a block and writing a block is simply not the way to go.

By reworking the problem as a concurrent problem (i.e., the reader and writer are both allowed to read and write blocks at whatever rate they can), performance can significantly be improved, thus addressing both aspects mentioned previously. It may not be obvious how the issue of “seek time” can be tamed by moving to a concurrent design. This will become apparent when you see (in the discussion and code) that it is possible for the reader to read (queue) several blocks from the source file before the writer is actually able to write the blocks to the destination file.

Before we turn to the details of the various classes used to implement concurrent file copying, the following list presents an overview of the classes and how they collaborate to implement the solution to the file-copying problem (we address each of these classes in detail later):

- **Buffer**: This class is an example of a wrapper class. Wrapper classes are often used in Java to make it easier to put data that belong together in a container. A `Buffer` is used to store a block of data that has been read from the source file and is to be written to the destination file.
- **Pool**: This class shows a useful pattern for minimizing the overhead associated with “garbage collection.” A `Pool` is simply a fixed collection of `Buffer` instances. These instances are borrowed from the `Pool` and returned as needed. This class actually makes use of conditions.
- **BufferQueue**: This class is used to maintain, in first-in-first-out (FIFO) order, a list of `Buffer` instances waiting to be written to the destination file. Unlike the `Pool`, any number of instances can appear on the `BufferQueue`. Conditions are also used by `BufferQueue` to provide flow control between the reader and the writer.
- **FileCopyReader1**: An instance of this class (which is a subclass of `Thread`) will read blocks from a file one at a time and write them to a `BufferQueue` that is shared with an instance of `FileCopyWriter1`. A `Pool` is also shared between an instance of this class and an instance of `FileCopyWriter1`.



- **FileCopyWriter1**: An instance of this class (which is a subclass of `Thread`) will read blocks from a `BufferQueue` instance and write them to the destination file. It is this class that is actually responsible for doing the final copying.
- **FileCopy1**: This is the main class that drives the entire application. This class is not used to create instances; it simply contains a `main()` method that processes the parameters and properties (we will talk about this shortly) and then creates the worker threads to copy a source file to a destination file.

This is intended to be an executive summary to help you understand at a high level how the code is actually assembled. We will now explore each of the classes in detail.

Example 3-7 shows the code for the `FileCopy1` class. This class contains the `main()` method and is used to drive the entire process of performing the file copy. Two command-line arguments are expected: the source and destination file names. Optionally, an “rc” file can be specified with a list of property bindings:

- **buffers**: These are the number of fixed buffers to be allocated *a priori* for file copying.
- **bufferSize**: This represents the number of characters (`char`) to be allocated per buffer.

Properties

`Properties` is a Java class that supports an environment mechanism (similar to environment variables) in a platform-independent manner.

`Properties` can be read from any stream (not just files). A property is defined as follows:
name=value

One property may be defined per line of input, separated by new lines. There is no limit on how many times a name can be used on the left-hand side.

There is more to the `Properties` class. It can inherit a set of default properties, so you never really have to hard code anything.

Once the command line arguments and properties have been processed, the real work begins. An instance of `Pool` (`pool`) is created, using the values `buffers` and `bufferSize` to do the allocation. An instance of `BufferQueue` (`copyBuffers`) is created to hold blocks that are to be copied to the destination file. Classes `Pool` and `BufferQueue` appear at first to be redundant; however, we emphasize they both have an important role in the design and are intentionally different to separate the concerns.



The `FileCopyReader1` and `FileCopyWriter1` classes are then instantiated (as `src` and `dst`, respectively) to perform the actual file copying. Note that `pool` and `copyBuffers` are shared object references being contained by both `src` and `dst` objects. Both objects are then started as threads; the `main()` thread simply awaits the completion of these two threads by performing a `join()` operation.

Example 3-7 FileCopy1: Multithreaded Version of FileCopy0

```
import java.io.*;
import java.util.*;

public class FileCopy1 {
    public static int getIntProp(Properties p, String key
                                int defaultValue) {
        try {
            return Integer.parseInt(p.getProperty(key));
        } catch(Exception e) {
            return defaultValue;
        }
    }

    public static void main(String args[]) {
        String srcFile = args[0];
        String dstFile = args[1];

        Properties p = new Properties();
        try {
            FileInputStream propFile =
                new FileInputStream("FileCopy1.rc");
            p.load(propFile);
        } catch(Exception e) {
            System.err.println("FileCopy1: Can't load Properties");
        }
        int buffers = getIntProp(p, "buffers", 20);
        int bufferSize = getIntProp(p, "bufferSize", 4096);
        System.out.println("source = " + args[0]);
        System.out.println("destination = " + args[1]);
        System.out.println("buffers = " + buffers);
        System.out.println("bufferSize = " + bufferSize);
        Pool pool = new Pool(buffers, bufferSize);
        BufferQueue copyBuffers = new BufferQueue();

        /* code continues on next page */
    }
}
```



Example 3-7 FileCopy1: Multithreaded Version of FileCopy0 (Continued)

```
/* continuation of main() method on previous page */

FileCopyReader1 src;
try {
    src = new FileCopyReader1(srcFile, pool, copyBuffers);
} catch(Exception e) {
    System.err.println("Cannot open " + srcFile);
    return;
}
FileCopyWriter1 dst;
try {
    dst = new FileCopyWriter1(dstFile, pool, copyBuffers);
} catch(Exception e) {
    System.err.println("Cannot open " + dstFile);
    return;
}
src.start();
dst.start();

try {
    src.join();
} catch(Exception e) {}
try {
    dst.join();
} catch(Exception e) {}
}
}
```

Example 3-8 shows the `Pool` class. This class is used as a storage manager and represents a useful technique for minimizing interactions with the garbage collector in Java programming. The `Pool` class employs the `Buffer` class (shown in Example 3-9), which is simply a wrapper class used both to maintain a reference to a `char []` and to keep track of the number of characters actually being used in the array. To understand why the `Buffer` class exists, recall that the copying is performed by using fixed-size blocks to do the reads and writes. More often than not, copying the very last block results in only a fraction of the actual character array being used. The `Buffer` class exists to elegantly handle this condition, as well as the end-of-file (EOF) condition. When EOF occurs, a `Buffer` of length zero is used to indicate the end-of-file boundary condition.

**Example 3-8 Pool: A Shared Pool of Buffer Structures**

```
import java.util.*;
import java.io.*;

public class Pool {
    Vector freeBufferList = new Vector();
    OutputStream debug = System.out;
    int buffers, bufferSize;

    public Pool(int buffers, int bufferSize) {
        this.buffers = buffers;
        this.bufferSize = bufferSize;
        freeBufferList.ensureCapacity(buffers);
        for (int i=0; i < buffers; i++)
            freeBufferList.addElement(new Buffer(bufferSize));
    }

    public synchronized Buffer use()
        throws InterruptedException {
        while (freeBufferList.size() == 0)
            wait();
        Buffer nextBuffer =
            (Buffer) freeBufferList.lastElement();
        freeBufferList.removeElement(nextBuffer);
        return nextBuffer;
    }

    public synchronized void release(Buffer oldBuffer) {
        if (freeBufferList.size() == 0)
            notify();
        if (freeBufferList.contains(oldBuffer))
            return;
        if (oldBuffer.getSize() < bufferSize)
            oldBuffer.setSize(bufferSize);
        freeBufferList.addElement(oldBuffer);
    }
}
```

The `Pool` class supports two key methods:

- **use:** This method is used to obtain a reference to the next available buffer. The method is always called by the reader thread (i.e., `FileCopyReader1`).



- **release:** This method is used to return an existing `Buffer` object to the `Pool`. This method is always called by the writer thread, after the contents of a given buffer have been written to the destination. No check is performed to determine whether the object originally came from the `Pool` (i.e., an honor system is assumed here); however, a check *is* performed to ensure that the object has a correct block size for the `Pool`. This requires that every `Buffer` object satisfies the invariant on `bufferSize`.

Example 3-9 Buffer

```
public class Buffer {
    private char[] buffer;
    private int size;

    public Buffer(int bufferSize) {
        buffer = new char[bufferSize];
        size = bufferSize;
    }

    public char[] getBuffer() {
        return buffer;
    }

    public void setSize(int newSize) {
        if (newSize > size) {
            char[] newBuffer = new char[newSize];
            System.arraycopy(buffer, 0, newBuffer, 0, size);
            buffer = newBuffer;
        }
        size = newSize;
    }

    public int getSize() {
        return size;
    }
}
```

The `BufferQueue` class (shown in Example 3-10) is used to maintain a list of `Buffer` objects in FIFO order. Since the `Vector` class of Java already supports FIFO functionality, the `BufferQueue` class can be construed as a wrapper class; however, it is also the

**Example 3-10 BufferQueue**

```
import java.util.*;
import java.io.*;

class BufferQueue {
    public Vector buffers = new Vector();

    public synchronized void enqueueBuffer(Buffer b) {
        if (buffers.size() == 0)
            notify();
        buffers.addElement(b);
    }

    public synchronized Buffer dequeueBuffer()
        throws InterruptedException {
        while (buffers.size() == 0)
            wait();
        Buffer firstBuffer = (Buffer) buffers.elementAt(0);
        buffers.removeElementAt(0);
        return firstBuffer;
    }
}
```

key class responsible for providing flow control between the reader and the writer. It is *also* the place where condition variables are used. There are two key methods:

- **void enqueueBuffer(Buffer b)**: This method puts *b* on the FIFO. If there is a thread waiting for a buffer to appear (because the FIFO is empty), this method notifies the thread.
- **Buffer dequeueBuffer()**: If the FIFO is empty, this method waits for the FIFO to become nonempty and removes the first buffer from the FIFO.

The `FileCopyReader1` and `FileCopyWriter1` classes (shown in Example 3-11 and Example 3-12, respectively, hereafter the “reader” and the “writer”) represent the rest of the story; the file copying is actually done by this pair of classes. Both of these classes work similarly. We will focus the remaining discussion on understanding the `run()` methods of both classes. The constructor code should be self-explanatory.



Example 3-11 FileCopyReader1

```
import java.io.*;
import java.util.*;

class FileCopyReader1 extends Thread {
    private Pool pool;
    private BufferQueue copyBuffers;
    private String filename;
    FileReader fr;

    public FileCopyReader1(String filename, Pool pool,
        BufferQueue copyBuffers) throws IOException {
        this.filename = filename;
        this.pool = pool;
        this.copyBuffers = copyBuffers;
        fr = new FileReader(filename);
    }

    public void run() {
        Buffer buffer;
        int bytesRead = 0;
        do {
            try {
                buffer = pool.use();
                bytesRead = fr.read(buffer.getBuffer());
            } catch (Exception e) {
                buffer = new Buffer(0);
                bytesRead = 0;
            }
            if (bytesRead < 0) {
                buffer.setSize(0);
            } else {
                buffer.setSize(bytesRead);
            }
            copyBuffers.enqueueBuffer(buffer);
        } while (bytesRead > 0);
        try { fr.close(); }
        catch (Exception e) { return; }
    }
}
```

**Example 3-12 FileCopyWriter1**

```
import java.io.*;
import java.util.*;

class FileCopyWriter1 extends Thread {
    private Pool pool;
    private BufferQueue copyBuffers;
    private String filename;
    FileWriter fw;

    public FileCopyWriter1(String filename, Pool pool,
        BufferQueue copyBuffers) throws IOException {
        this.filename = filename;
        this.pool = pool;
        this.copyBuffers = copyBuffers;
        fw = new FileWriter(filename);
    }

    public void run() {
        Buffer buffer;
        while (true) {
            try {
                buffer = copyBuffers.dequeueBuffer();
            } catch (Exception e) { return; }
            if (buffer.getSize() > 0) {
                try {
                    char[] buffer = buffer.getBuffer();
                    int size = buffer.getSize();
                    fw.write(buffer, 0, size);
                } catch (Exception e) {
                    break;
                }
                pool.release(buffer);
            } else break;
        }
        try { fw.close(); }
        catch (Exception e) { return; }
    }
}
```



The `run()` method of the reader works as follows:

- It obtains a `Buffer` instance, `buffer`, from the `Pool of Buffers`.
- It reads up to `buffer.getBuffer().length` bytes from the source file (stream). Note that `getBuffer()` is returning a reference to the contained array (`char[]`) object and that the `read()` call works with arrays. (In Java, unlike C, this operation can be performed safely, since the length of an array is always known and cannot be violated.)
- If, for any reason, an exception occurs [I/O is most likely at the point of the `read()` call], `run()` creates a buffer of size 0.
- If EOF occurs (which is not an exception in Java), sets the size of `buffer` to zero. This will allow the writer to know that EOF has been encountered and terminated gracefully.
- Otherwise, `run()` sets the size of `buffer` to `bytesRead`, the number of bytes actually read. This number is always guaranteed to be less than `buffer.getSize()`.
- It enqueues `buffer` onto `copyBuffers`.

The `run()` method of the writer is very similar to the reader code and works as follows:

- It dequeues a `buffer` from `copyBuffers`. This is the next block of data to be copied.
- If `buffer.getSize() > 0`, we have not yet reached the end of file. `run()` then writes the block to the file. If end of file has been reached, the program exits the thread.
- It returns `buffer` to `pool`. This is very important to do, as the reader could be waiting for the pool to be nonempty.

Locks—Binary Semaphores: An Example of Using Conditions

Other multithreading systems do not have implicit locks built into the syntax, but instead require you to create and call methods of explicit *lock* objects. A lock object is also called a mutex or a binary semaphore. A given lock may only be in one of two states: locked or unlocked. Initially, a lock is normally in the *unlocked* state. Example 3-13 presents the code for an explicit lock object, written in pure Java.

The following relatively intuitive operations are defined on locks:

- `lock()`—This operation locks the lock. The lock is first tested (atomically) to see whether it is locked. If it is locked, the current thread must wait until it becomes unlocked before proceeding.

**Example 3-13 Lock: A Class to Support Binary Semaphores**

```
// Lock.java - a class to simulate binary semaphores

class Lock {
    protected boolean locked;

    public Lock() {
        locked=false;
    }

    public synchronized void lock()
        throws InterruptedException {
        while (locked) wait();
        locked=true;
    }

    public synchronized void unlock() {
        locked=false;
        notify();
    }
}
```

- `unlock()`—This operation unlocks the lock. If the lock is currently in the locked state, `unlock` notifies any thread that may be waiting and the state of the lock is then set to unlocked.

Example 3-13 shows the code for a Java implementation of the concept of a lock, which makes use of conditions.

Notice that the code follows almost exactly from the above definitions. The state of the lock is represented by a boolean variable. This variable is set true if the lock is presently locked and false otherwise. The constructor for class `Lock` initializes `locked` to false. It is perfectly acceptable to have a constructor that initializes `locked` to true; however, this is not commonly practiced in real-world applications. (For another kind of lock, called a semaphore—which maintains a count—such an initialization is meaningful. We will discuss this possibility a bit later.)

The `lock()` method checks to see whether the state of the lock is currently locked. Notice that, as described earlier, a `while` loop is used to test this condition. This loop will not terminate unless the state of the lock becomes unlocked. The `unlock` method explicitly



sets the state of the lock to false (without checking its current state) and subsequently notifies any waiting thread that the condition has changed.

Typically, a thread that performs the lock operation is going to perform a matching unlock operation (on the same lock) after executing its critical section; however, it is not an absolute requirement that a lock be used in this manner. (In fact, to impose this restriction would complicate the lock implementation considerably!) Therefore, it is possible that a group of threads could all have references to a common lock and use it in a rather undisciplined way. For instance, one thread could just perform an unlock while another thread “thought” it held the lock. Locks are intended to be used in a disciplined way (i.e., following the prescribed steps). Java enforces lock discipline by not allowing the possibility of a user ever locking an object without eventually unlocking it in a given block of code (short of intentionally setting up an infinite loop).

The `throws InterruptedException` declaration deserves a brief explanation. Because the `lock()` method calls `wait()`, it can result in an `InterruptedException`. It simply lets that exception pass through to its caller, so `InterruptedException` must be declared in the `throws` clause. This is a very common programming practice in Java. When working with exceptions in Java, you must either handle the exception right away or pass the exception to an outer context.

Because the `Lock` class defined here is intended to be useful to those in need of an explicit lock similar to those found in other threads packages (usually used by C and C++ programmers), the latter approach has been taken. Users will expect the behavior of `lock()` to be similar to that of `wait()`, since both have the potential of blocking and being subsequently interrupted.

Locks: Where else can you find them?

Locks are found in both the `pthread`s and `Win32` threads libraries. In `pthread`s, you find `pthread_mutex_t`, which is the type used to declare a lock variable.

A number of methods exist to actually manipulate this variable:

`pthread_mutex_init()`—This method is used to initialize the `mutex` with a set of attributes. These attributes can be specified using a variable of type `pthread_mutexattr_t`.

`pthread_mutex_lock()`—This method is used to perform the `lock` operation on the `mutex`.

`pthread_mutex_unlock()`—This method is used to perform the `unlock` operation.

The `Win32` library works similarly:

`HANDLE`—This method is used to refer to the lock.

`HANDLE CreateMutex()`—This method is used to create a `mutex` and return a `HANDLE`.

`WaitForSingleObject()`—This method is used to perform the `lock` operation on the `mutex`.

`ReleaseMutex`—This method is used to perform the `unlock` operation on the `mutex`.



Race2: Reworked Race1 Using Locks

Example 3-14 shows the code used to eliminate the race condition found in `Shared0`. The code employs the `Lock` class we just developed. As before, with the examples of `Race1` and `Shared1`, significant changes are required only for the `Shared` class—the class that represents the shared data to be protected with synchronization. A member variable, `mutex`, refers to an instance of the lock used to protect the other variables `x` and `y` whenever `dif()` and `bump()` are called.

Note that, in this example, `dif()` and `bump()` are not synchronized, because the synchronization will be done entirely in `mutex.lock()` and `unlock()` calls. These calls are both themselves declared to be *synchronized*. They are now also declared to throw an `InterruptedException`, since they both call `lock()` on `mutex`, which can throw the exception, and then they simply pass it back to their caller.

An interesting difference between the implementation of `dif()` in class `Shared2` and `dif()` in `Shared0` or `Shared1` is that `dif()` in `Shared2` needs to declare a temporary variable, compute the difference, and then return the temporary variable. This might appear awkward, but it is easily understood by considering that the synchronized version of `dif()` (as presented in `Shared1`) does not need to worry about locking and unlocking because both are done implicitly. Thus, when the `return x-y;` command is encountered in a synchronized block, the `return` statement causes the object's lock to be

Example 3-14 `shared2`: Eliminating Race Conditions Using Lock Class

```
class Shared2{
    protected int x=0,y=0;
    protected Lock mutex=new Lock();
    public int dif() throws InterruptedException {
        int tmp;
        mutex.lock();
        tmp=x-y;
        mutex.unlock();
        return tmp;
    }
    public void bump() throws InterruptedException {
        mutex.lock();
        x++;
        Thread.sleep(9);
        y++;
        mutex.unlock();
    }
}
```



released. The same is not true of the `dif()` method provided in `Shared2`, because the method is not declared synchronized. Thus, there is no applicable lock for the `Shared2` instance itself.

It is possible to simulate what is normally done in Java by making use of the `try` statement. This statement is usually used to execute one or more statements that may result in an exception. Additionally, the statement can be used to guarantee that something is done, regardless of the outcome of the statements, by using the `finally` clause. The `dif()` code can be modified as shown in Example 3-15. It is probably best to put the unlocking code in `finally` clauses. That is what the Java compiler generates for synchronized statements and methods.

In general, explicit locks (such as the `Lock` class) should not be used as an alternative to the Java `synchronized` keyword. The `Lock` class has been illustrated here because many programmers who have worked with concurrency in other languages (such as C and C++) will undoubtedly be familiar with it. The `Lock` class is also an excellent (minimal)

Example 3-15 Reworking `Shared2` to Guarantee Proper Release of the Explicit Lock.

```
class Shared2{
    protected int x=0,y=0;
    protected Lock mutex=new Lock();
    public int dif() throws InterruptedException {
        mutex.lock();
        try {
            return x-y;
        } finally {
            mutex.unlock();
        }
    }
    public void bump() throws InterruptedException {
        mutex.lock();
        try {
            x++;
            Thread.sleep(9);
            y++;
        } finally {
            mutex.unlock();
        }
    }
}
```



example of how to use conditions. Should you ever forget how a condition in Java works, feel free to revisit the `Lock` class as a refresher example.

Is the `Lock` Class Useful?

The answer to this question depends on your point of view. Java objects only provide you with a single lock. Thus, if you need multiple locks to protect variables separately, you would need to have separate objects. This is fairly easy to do: Simply declare a variable of type `Object`, and make sure that it really refers to an object:

```
Object xLock = new Object();
```

Then use a synchronized block to protect the critical section:

```
synchronized (xLock) {  
    }  
}
```

Of course, the cost of creating an instance of class `Lock` is the same and is perhaps easier to use. It's up to you!

Classic Synchronization Mechanisms

The `Lock` class illustrated how conditions can be useful to evolve a familiar synchronization mechanism using the intrinsic lock and condition mechanisms supported by every Java object. The remainder of this chapter is dedicated to other high-level synchronization mechanisms: counting semaphores, barriers, and simple futures. Counting semaphores are addressed in numerous textbooks on operating systems and systems programming. Barriers are discussed in textbooks on parallel processing and are used extensively in scientific computation. Futures have their roots in functional programming and have been discussed in numerous textbooks on programming languages.

Counting Semaphore

A counting semaphore is very similar to a lock. Recall that a lock is sometimes called a binary semaphore. This is because it is either in a locked (1) or an unlocked (0) state. In the counting semaphore, an integer count is maintained. Usually, the semaphore is initialized to a count `N`. There are two operations that are defined on the semaphore:

- **up()**: This operation adds one to the count and then notifies any thread that may be waiting for the count to become positive.
- **down()**: This operation waits for the count to become positive and then decrements the count.

Example 3-16 shows the code for a counting semaphore.

The implementation of a counting semaphore is very similar to that of a lock. The first noticeable difference between the two is that the methods are named `down()` and `up()` instead of `lock()` and `unlock()`. The reason for this difference is that semaphores are usually defined in the former terms in typical operating systems textbooks. (The original



Are Counting Semaphores Useful?

Counting semaphores are very useful. They can be used to allow at most N resources to be utilized at the same time. A resource can be anything you want it to be. Operating-systems textbooks usually give examples of resources that are managed by an OS, such as disk drives, tape drives, etc. Other examples of resources (in the context of this book) are open network connections, worker pools, etc.

Sometimes, it is easy to overlook the obvious. The semaphore can easily be used to maintain a safe atomic counter between a group of threads. Of course, this could be construed as a case of overkill, since atomic counters are supported by the Java standard using volatile variables. Any variable marked volatile cannot be cached and therefore has a write-through update policy, used when any operation is performed on it. An example of the code is

```
volatile int x = 0;
x++;
```

Example 3-16 Class Semaphore: An Implementation of a Counting Semaphore

```
public class Semaphore {
    protected int count;

    public Semaphore(int initCount)
        throws NegativeSemaphoreException{
        if (initCount<0)
            throw new NegativeSemaphoreException();
        count=initCount;
    }
    public Semaphore() { count=0; }

    public synchronized void down()
        throws InterruptedException {
        while (count==0) wait();
        count--;
    }
    public synchronized void up() {
        count++;
        notify();
    }
}
```

paper on the subject, written by Dijkstra, actually used $P()$ and $V()$, the first letters of the Dutch words for “signal” and “notify,” respectively.)

The `down()` method implementation resembles that of the `lock()` method. It waits indefinitely for the semaphore count to be positive. When the count becomes positive,



the count will be decremented. As done in the `lock()` method (class `Lock`), the `InterruptedException` is passed to an outer context by declaring `throws InterruptedException` in the method header. The `up()` method is very similar to the `unlock()` method (class `Lock`). The counter is incremented and notifies any waiting thread that the count has a positive value.

Barrier

In parallel and distributed computing applications, there is often a need to synchronize a group of threads (parallel tasks), usually inside a loop. A later chapter will address loop parallelism in greater detail. A barrier is an abstraction for performing this synchronization and is fairly straightforward. It is similar to the notion of a semaphore, but works somewhat backwards. The barrier, much like the counting semaphore, is initialized to a count `N`. The `barrier()` method, when called by a thread, decrements the count atomically. Then the caller waits (is blocked) until the barrier count reaches 0. If any call to `barrier()` results in the count reaching 0, all threads awaiting the count of 0 are notified, and the count is reset to `N`.

Thus, the `barrier()` is a great deal different than a semaphore in two ways:

- Most threads (`N-1`) reach the barrier and are forced to wait until the count is correct. When calling `down()` on a semaphore, a thread waits only when the count is 0.
- All threads are notified when the count reaches 0. In a semaphore, only a *single* thread is notified each time the `up()` operation is called.

Example 3-17 shows the code for the implementation of class `Barrier`.

The `Barrier` class has two instance variables:

- **count**: This variable is the current value of the barrier. When `count` becomes 0, it is reset to the value of variable `initCount`.
- **initCount**: This variable is the initial value of the barrier. It is a cached copy of the value passed in the constructor.

The constructor initializes `Barrier` to a positive count. An initial value that is negative or 0 results in a `BarrierException` code being thrown. (For conciseness, the code for `BarrierException` is not shown here, but it can be found on the accompanying CD. Its details are not essential to understand the `Barrier` class.) The `count` and `initCount` fields are initialized to the specified count.

The `barrier` method atomically decrements the count. If the count has not yet reached 0, the caller blocks indefinitely; otherwise, the count is reset to the value specified originally in the `Barrier` constructor, and all waiting threads are notified and, hence, unblocked.



Example 3-17 Barrier Synchronization

```
// Barrier.java

public class Barrier {
    protected int count, initCount;

    public Barrier(int n) throws BarrierException{
        if (n <= 0)
            throw new BarrierException(n)
        initCount = count = n;
    }

    public synchronized void barrier()
        throws InterruptedException {
        if (--count > 0)
            wait();
        else {
            count = initCount;
            notifyAll();
        }
    }
}
```

An interesting question now arises: Earlier in the chapter, we mentioned that the proper way to wait for a condition is to first use a `while` loop to test the condition and then wait. In the `barrier()` method, this has not been done. The reason is that we are not waiting for the barrier to reach any particular value; we are simply waiting for it to be reset. Once it has been reset, every single thread that posted a `wait()` should be awakened and allowed to proceed. Effectively, the last thread to make a call to the `barrier()` method does not wait. Only the first $N-1$ threads to arrive at the `barrier()` wait.

Futures

A future is an assign-once variable that can be read any number of times once it is set. Futures have their origins in functional programming languages. A typical use for a future is to support the familiar notion of a procedure call, in which the arguments being passed to the call may not yet have been evaluated. This is often acceptable, since very often a parameter is not used right away in a procedure call. (It is very difficult to use all of the parameters simultaneously. This fact can be verified by inspecting a substantial piece of



code with a few parameters.) Futures also make sense in a multiprocessor system, in which there may be many processors to evaluate a set of futures.

A future is implemented in Java using a “wrapper.” The code for the class `Future` is shown in Example 3-18. The wrapper is used to maintain a protected reference to a value. If the future has not yet been set, the value is set to the future instance itself. (This is a sensible value, since a future cannot have itself as a value, but must be able to have the value null. Alternatively, a boolean flag could be maintained; however, as Example 3-18 points

Example 3-18 The Future

```
public class SimpleFuture {
    protected Object value;

    public SimpleFuture() {value=this;}

    public SimpleFuture(Object val) {
        value=val;
    }

    public synchronized Object getValue()
        throws InterruptedException{
        while (value == this)
            wait();
        return value;
    }

    public synchronized boolean isSet(){
        return (value!=this);
    }

    public synchronized void setValue(Object val){
        if (value == this) {
            value = val;
            notifyAll();
        }
    }
}
```



out, some of the ugliness that is inherent in the Java object model happens to be useful!) There are two key methods for working with a future:

- **getValue()**: This method is used to wait until the future has been set, then return a reference to the value of the future.
- **setValue()**: This method is used to set the value of the future, if it has not already been set. It also notifies all threads that might be awaiting the value of the future. If the future is already set, its value cannot be changed, and no further notifications are to be done.

Futures will be addressed again in greater detail later in this book. For now, think of a future as a clever trick used to wait for something to be produced—once. The concept of a future could be used, for example, to implement the concurrent-Web-page example, discussed in Chapter 1. For each Web page to be read, a thread can be created. Once the page has been read, it can be “written” to a future. Then, another waiting thread (usually, the main one) can wait for all of the futures, one at a time. The interesting aspect of this is that one can write concurrent applications that are fairly sophisticated without having to know all of the details of how threads and synchronization really work. Of course, we will discourage you from not knowing the details, since we want you to know them.

Deadlock

What is Deadlock?

Deadlock occurs when each thread in a group of threads is waiting for a condition that can only be caused by another member of the the group. For example, given threads T1 and T2, it is possible that T1 is holding a resource X while waiting for resource Y to become available. Meanwhile, T2 is holding resource Y and is waiting for resource X to become available. This condition is known, in the computing literature, as deadlock.

How to Know When Deadlock Has Hit You?

Aside from having to debug a distributed program (i.e., one that runs on many machines—something we will discuss later in this book), deadlock is perhaps one of the most difficult programming errors to correct. At times, deadlock appears to behave almost like an infinite loop. Sometimes, the program will appear to run normally by not “locking up.” Often, the program will need to be run multiple times to even notice the problem. The tragedy of it all is that the results are usually not reproducible nor consistent.

Four Conditions of Deadlock

Four conditions enable the possibility for deadlock:

- **Mutual exclusion:** It must not be possible for more than one thread to use a resource at the same time.



- **Hold and wait:** Threads must hold resources and wait for others to become available. Deadlock is not possible if no thread ever holds more than one resource at a time. Nor is it possible if a thread can acquire more than one resource, but acquires all its resources at one instant.
- **No preemption:** It must not be possible to remove a resource from a thread that holds it. Only the thread that holds a resource can give it up.
- **Circular wait:** There must be a cycle of threads, each holding at least one resource and waiting to acquire one of the resources that the next thread in the cycle holds.

If any of the foregoing conditions is enforced, deadlock simply cannot occur. This idea comes from operating systems literature (textbooks). We will show how to put it into practice, focusing on eliminating circular waits.

A Classic Example: Dining Philosophers

Synchronization problems have been the subject of study for many decades. One of the classic problems, known as the Dining Philosophers problem, was presented by Dijkstra, who is also credited with the invention of the semaphore (which was presented earlier).

The Dining Philosophers problem is stated as follows. There are five philosophers sitting around a circular dinner table eating spaghetti. (You can substitute your favorite pasta here, without affecting your overall understanding of the problem.) Each philosopher is in one of two states: thinking or eating. Each philosopher has his own plate. There are five forks, one located between each plate. To eat, a philosopher needs two forks.

After thinking for a while, the philosopher will stop thinking, pick up one fork at a time, one from each side of his plate, and eat for a while. Then he will put down the forks, again one at each side of his plate, and return to thinking.

If each philosopher starts to eat at the same time, each might pick up the fork from the left side of his plate and wait for the one to the right to become available, resulting in deadlock. To solve the problem, a Java program that avoids such deadlocks needs to be written.

The code is organized in three classes:

- **Fork:** This class is an object that is shared between a pair of diners (instances of `Diner0`; see next).
- **Diner0:** This class is a threaded object that implements the moves of a typical diner in its `run()` method.
- **Diners0:** This class is a driver code that simply creates instances of `Fork` and `Diner0` and then sets the simulation in motion.

A `Fork` (shown in Example 3-19) is implemented using the `Lock` class described in Example 3-13. For output purposes, an `id` is maintained for each `Fork`. Aside from this difference, a fork, for all intents and purposes, is a lock. In keeping with our desire to be



Example 3-19 Fork

```
class Fork {
    public char id;
    private Lock lock=new Lock();

    public void pickup() throws InterruptedException {
        lock.lock();
    }

    public void putdown() throws InterruptedException {
        lock.unlock();
    }

    public Fork(int i) {
        Integer i = new Integer(i);
        id = i.toString().charAt(0);
    }
}
```

object oriented, the methods are named `pickup()` and `putdown()`, which are the actual operations that can be performed on a fork (*responsibilities*, in object jargon).

Each diner is represented by `Diner0` (shown in Example 3-20). A diner has a state (which again exists for the purpose of output) and references to two forks (L and R in the code). The `run()` method of the diner is where the diner actually does his work. He thinks (which blocks him for a specified time), then picks up the left fork L, blocks himself, then picks up his right fork R, then eats (again, blocking himself), and then puts both forks L and R down to take a break from eating. This process is repeated 1,000 times for each diner.

The driver code (shown in Example 3-21) is responsible for setting the table, metaphorically speaking. This is done in the `main()` method. Two arrays are initialized:

- **fork**: This is an array of `Fork` instances.
- **diner**: This is an array of `Diner` instances.

The assignment of forks to diners is relatively straightforward. `Diner 0` is assigned forks 0 and 1; `diner 1` is assigned forks 1 and 2, and so on. The last diner, 4, is assigned forks 4 and 0. It will be apparent that this last numbering, although it makes the coding somewhat easier by not having to encode a special case, is precisely what causes the possibility of a deadlock.

Once all of the forks and diners have been created, the simulation of dining is commenced. The `main()` method, being effectively the main thread, establishes itself as a “watcher”

**Example 3-20 Diner0**

```
class Diner0 extends Thread {
    private char state='t';
    private Fork L,R;

    public Diner0(Fork left, Fork right){
        super();
        L=left;
        R=right;
    }

    protected void think() throws InterruptedException {
        sleep((long)(Math.random()*7.0));
    }

    protected void eat() throws InterruptedException {
        sleep((long)(Math.random()*7.0));
    }

    public void run() {
        int i;
        try {
            for (i=0;i<1000;i++) {
                state = 't';
                think();
                state=L.id;
                sleep(1);
                L.pickup();
                state=R.id;
                sleep(1);
                R.pickup();
                state='e';
                eat();
                L.putdown();
                R.putdown();
            }
            state='d';
        } catch (InterruptedException e) {}
    }
}
```

or “monitor” of the other threads. Its priority is set to be slightly higher than any of the other running threads.



Example 3-21 Diners0: The Driver

```
class Diners0 {
    static Fork[] fork=new Fork[5];
    static Diner0[] diner=new Diner0[5];

    public static void main(String[] args) {
        int i,j=0;
        boolean goOn;

        for (i=0;i<5;i++) {
            fork[i]=new Fork(i);
        }
        for (i=0;i<5;i++) {
            diner[i]=new Diner0(fork[i],fork[(i+1)%5]);
        }
        for (i=0;i<5;i++) {
            diner[i].start();
        }
        int newPrio = Thread.currentThread().getPriority()+1;
        Thread.currentThread().setPriority(newPrio);
        goOn=true;
        while (goOn) {
            for (i=0;i<5;i++) {
                System.out.print(diner[i].state);
            }
            if (++j%5==0)
                System.out.println();
            else
                System.out.print(' ');
            goOn=false;
            for (i=0;i<5;i++) {
                goOn |= diner[i].state != 'd';
            }
            try {
                Thread.sleep(51);
            } catch (InterruptedException e) {return;}
        }
    }
}
```

The main thread then goes into an indefinite (not infinite) loop, which will terminate only when all of the diners are in the “done” state. The only way this can happen is if a given diner thread terminates (i.e., executes its loop 1,000 times). The way thread termination is determined is by examining the *state* of each particular diner to see whether or not it is d



(done). The only way `goOn` (the condition by which the monitor thread's loop is terminated) becomes false is when all of the diners are in the `d` state.

Taking a step back, we now want to know what is the point of setting the priority? Java uses a priority scheduler that will run the highest priority thread available to run, or one of the highest priority threads available if there is more than one at the highest priority level. The reason we set the priority level of the main thread higher than that of the diners is so that any running diner can be preempted to examine (and print) its state.

There are only a few priority levels available to a thread (12), which are represented as an integer. The higher the integer, the higher the priority. You can examine a thread's priority using method `getPriority()`, and assign a thread a new priority using `setPriority()`. As mentioned in the previous chapter, there are also a number of constant values available in the `Thread` class to make use of normal priority values: `Thread.MAX_PRIORITY`, `Thread.NORMAL_PRIORITY`, and `Thread.MIN_PRIORITY`. You must be careful to ensure that a higher priority thread does not prevent other threads from running. This is always a possibility, especially when you use `MAX_PRIORITY`. Until the thread does something to block itself (I/O or synchronization—two examples of a blocking primitive), the thread will be allowed to run to completion, preventing any other threads from running in the process. In this example, the watcher thread is explicitly putting itself to sleep for so many milliseconds—hence the `Thread.sleep(51)`. This is intended not as a kludge, but to explicitly guarantee that the main thread does block for a reasonable amount of time. Remember this: The main thread's purpose for existence is to show you the states of the various diners so you will know how much progress they are making as a whole (and so you will see what happens when they get into a deadlock situation).

The following box shows output from a particular run. As discussed for the example `Race0`, the output is likely to be different between different runs, due to different serializations of events.

A brief explanation of the output is in order. The output represents the states of the five diners in order (0 through 4). The output is grouped by iteration in the main thread, separated by either a space or a new line, the intent being to make the output as neat as possible without using too many lines. Of course, because the program occasionally deadlocks, this causes an infinite loop to occur—the last line of output clearly shows that a circular wait has occurred. That is, each diner is waiting for a fork that he will never be able to acquire. (i.e., never be able to *pick up*).

As mentioned earlier, there are four ways in which deadlock can occur. To eliminate deadlock, all you need to do is find a way to eliminate one of the conditions that can cause it. It is not always desirable to eliminate a given condition. We will return to a discussion of this point shortly; however, we consider here the possibility of eliminating a *circular wait* condition.



```
1133t 0e24e 02t30 023et 0t24e
0e24e t2230 02t30 t234e et240
12et4 t134e t134e 1et30 023et
023e4 et240 1et34 023et t134e
e1240 12et4 t134e 1et30 02e30
023et t124t 0tt4e 12e30 t134e
1et30 12et4 113et t134e 1et30
1tt34 02et0 t13et e1240 1e230
12et4 t234e tt230 1et34 02e34
t134e 1e230 02e30 t23e4 t2ett
e1340 1et30 023et t134e 1et30
023et t134e 1et30 12e34 12et4
t134e ttt3e et230 1et34 023et
et240 12et4 123e4 0234e t1240
1et30 023et 023et et340 12e34
02t44 0134e et240 12e34 013tt
et240 01240 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340
```

In the above run, it is clear where the deadlock occurs. The simulation gets stuck in a pattern “12340,” which indicates that a circular wait condition has occurred that will never terminate. In this simulation, execution was halted explicitly by typing *control-C*.

One way to prevent circular waiting is to do *resource enumeration* and only allow individual threads to acquire resources in ascending order.

When you acquire resources (such as the `Fork` object) in ascending order, circular wait is impossible: for a circular wait to occur, some thread must be holding a higher numbered resource waiting for a lower numbered resource, but that is forbidden. Here is how we can use resource enumeration in the Dining Philosophers problem to eliminate deadlock.

The following loop excerpt from Example 3-21 to define a `Diner` is where the problem occurs:

```
for (i=0;i<5;i++) {
    diner[i]=new Diner0(fork[i],fork[(i+1)%5]);
}
```

A simple fix, which reorders the `Fork` objects such that a lower-numbered instance is always picked up first is to do the following:

```
for (i=0;i<4;i++) {
    diner[i]=new Diner0(fork[i],fork[(i+1)%5]);
}
diner[4] = new Diner0(fork[0], fork[4]);
```



It should be obvious how this code fixes the problem. The problem is caused by the last iteration of the loop, which is equivalent to the following long-winded code (an expansion of the first version of the loop):

```
diner[0] = new Diner0(fork[0], fork[1]);
diner[1] = new Diner0(fork[1], fork[2]);
diner[2] = new Diner0(fork[2], fork[3]);
diner[3] = new Diner0(fork[3], fork[4]);
diner[4] = new Diner0(fork[4], fork[0]);
```

The second version of the loop stops one short of the last iteration and simply does an explicit ordering of the final two forks. Instead of `fork[4]` being diner 4's left fork and `fork[0]` being diner 4's right fork, the forks are swapped. The notion of left and right is purely a matter of convention: one of the philosophers can always be considered to be left handed.

In summary, the diners are programmed to pick up their left fork before their right fork. So `diner j`, for `j` ranging from 0 through 3, will pick up `fork j` and then `fork j+1`. We explicitly tell diner 4 to pick up fork 0 before fork 4. The reworked code with the above changes appears in Example 3-22. The changes to support resource enumeration are shown in bold.

Chapter Wrap-up

This chapter has focused on race conditions, which represent one of your biggest enemies when it comes to programming concurrent applications. Java helps you to address the problem of race conditions using object locking, which is supported by each and every Java object created using the `new` operator. Java is one of the first modern (and popular) languages to support object locking. In languages such as C and C++, programmers are forced to make use of platform-specific libraries, such as `pthread`s (the UNIX standard) and `Win32 threads` (the Microsoft de facto standard). This by no means implies that a great amount of work has not been done to bring similar features to C and C++; however, in all cases and at some level, the solution is predominantly driven by libraries, not by the language itself.

While Java makes it considerably easier to support concurrency by providing support directly in the language, its features, in many respects, are not radically different from those found in `pthread`s or `Win32 threads`. Often, programmers find the need for more elaborate synchronization than that provided by merely locking objects. Java conditions were discussed as a way of providing more elaborate synchronization—a lock can be held for a while, but released “temporarily” while awaiting a certain condition. This can lead to better performance, since repeatedly acquiring and releasing locks does have a cost. As will be discussed in the next chapter, the word “wait” introduces the possibility of an encounter with another common enemy: deadlock.



Example 3-22 Diners1: The Driver

```
class Diners1 {
    static Fork[] fork=new Fork[5];
    static Diner0[] diner=new Diner0[5];

    public static void main(String[] args) {
        int i,j=0;
        boolean goOn;

        for (i=0;i<5;i++) {
            fork[i]=new Fork(i);
        }
        for (i=0;i<4;i++) {
            diner[i]=new Diner0(fork[i],fork[(i+1)%5]);
        }
        diner[4]=new Diner0(fork[4],fork[0]);

        for (i=0;i<5;i++) {
            diner[i].start();
        }
        int newPrio = Thread.currentThread().getPriority()+1;
        Thread.currentThread().setPriority(newPrio);
        goOn=true;
        while (goOn) {
            for (i=0;i<5;i++) {
                System.out.print(diner[i].state);
            }
            if (++j%5==0)
                System.out.println();
            else
                System.out.print(' ');
            goOn=false;
            for (i=0;i<5;i++) {
                goOn |= diner[i].state != 'd';
            }
            try {
                Thread.sleep(51);
            } catch (InterruptedException e) {return;}
        }
    }
}
```

The chapter concluded with a discussion of some other mutual-exclusion and synchronization mechanisms that are widely familiar to programmers—locks, semaphores, and bar-



riers—and one that perhaps is less familiar: the future. These classes are all built rather easily from the concurrency framework provided by Java and are very useful for developing concurrent applications. (One could argue that these classes belong in the `java.lang` package because they are foundational, but this is an argument for another day and time.)

Exercises

1. (Easy) Explain what the following code from Example 3-2 is doing:

```
System.out.print("X".charAt(s.dif()));
```

In particular, explain why it is permissible for a string “constant” to appear on the left-hand side of the dot operator. Is there ever the possibility that this string is not indexed properly with the `s.dif()` call?
2. (Easy) In the same example as the preceding problem, what are all of the valid results of `s.dif()`? Sketch a proof of why it is impossible for a value other than one of these results to be produced.
3. (Easy) Using the Java documentation, see if you can find classes that make use of synchronized methods. What classes can be used in a thread-safe manner? Can you find a Java class that might not work reliably in a multithreaded context?
4. (Intermediate) Sketch or implement a program that will compute the length of a time slice in milliseconds. If it is not possible to develop such a program, provide an explanation.
5. (Intermediate) Study the Java library source code to better understand the classes you discovered in Part 3.
6. (Intermediate) In Example 3-1, why is it necessary for the `q.setDaemon(true)` method call to be inserted in the main method? Is there another way to gracefully prevent the main method from exiting prematurely?
7. (Intermediate) Explain why it is possible for two (or more) threads to enter a synchronized method (or methods) when one (or more) threads is waiting for a condition. As a more advanced problem, explain what is happening from the operating system’s point of view.
8. (Easy) `FileCopy0` and `FileCopy1` are both working versions of a concurrent file-copying algorithm. Using the `System` class (which provides support for “timing” using the `currentTimeMillis()` method), compare the performance of the two programs. Use a sufficiently large file (at least one megabyte) with a block size of 4,096 to do the comparison.
9. (Easy) If a `RuntimeException` occurs in the `run()` method (e.g., something like a `NullPointerException`), does the method terminate?



10. (Intermediate) Example 3-2 shows a version of shared data that does not make use of synchronized methods. Is synchronization necessary in this case? Why or why not?
11. (Easy) The Semaphore class shown in Example 3-16 is an example of how to implement a counter that can be incremented or decremented atomically, provided its value never becomes negative. Show how to implement a class called Counter that can be incremented or decremented at will by a particular value.
12. (Intermediate) Compare the locking framework of Java (the synchronized statement) with the Lock and Semaphore classes shown in this chapter. Does Lock do anything that synchronized does not? What about Semaphore? How do Lock and Semaphore compare with one another?
13. (Hard) Using the URL class of Java, write a program that fetches three Web pages in a loop. Use the timing function (see problem 5) of the System class to determine the time. Write a second version of this program that creates separate threads to fetch each page and uses an array of Future instances. The main thread should wait for each Future instance. Again, use the timing function to determine the total execution time.