# CHAPTER 4

# Threads

The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Most modern operating systems now provide features enabling a process to contain multiple threads of control. This chapter introduces many concepts associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Win32, and Java thread libraries. We look at many issues related to multithreaded programming and its effect on the design of operating systems. Finally, we explore how the Windows and Linux operating systems support threads at the kernel level.

## CHAPTER OBJECTIVES

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries.
- To examine issues related to multithreaded programming.

## 4.1 Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or **heavyweight**) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 4.1 illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

### 4.1.1 Motivation

Many software packages that run on modern desktop PCs are multithreaded. An application typically is implemented as a separate process with several threads of control. A Web browser might have one thread display images or

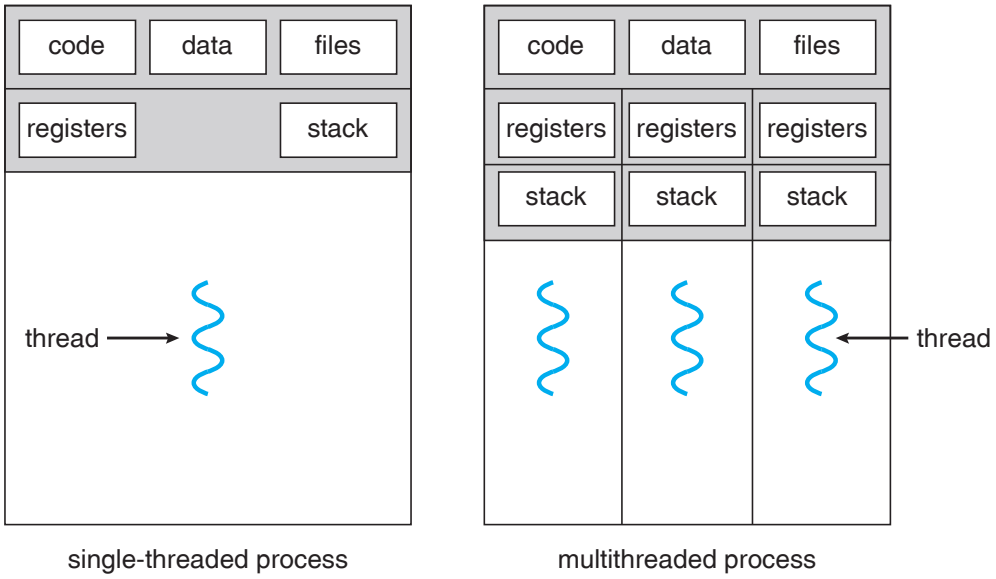


**Figure 4.1** Single-threaded and multithreaded processes.

text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations, a single application may be required to perform several similar tasks. For example, a Web server accepts client requests for Web pages, images, sound, and so forth. A busy Web server may have several (perhaps thousands of) clients concurrently accessing it. If the Web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the Web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server will create a new thread to service the request and resume listening for additional requests. This is illustrated in Figure 4.2.

Threads also play a vital role in remote procedure call (RPC) systems. Recall from Chapter 3 that RPCs allow interprocess communication by providing a communication mechanism similar to ordinary function or procedure calls. Typically, RPC servers are multithreaded. When a server receives a message, it services the message using a separate thread. This allows the server to handle several concurrent requests.

Finally, most operating system kernels are now multithreaded: several threads operate in the kernel, and each thread performs a specific task, such

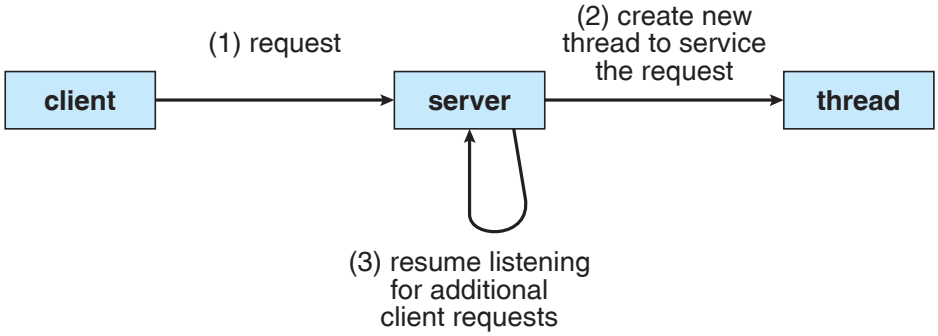


**Figure 4.2** Multithreaded server architecture.

as managing devices or interrupt handling. For example, Solaris creates a set of threads in the kernel specifically for interrupt handling; Linux uses a kernel thread for managing the amount of free memory in the system.

### 4.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.
2. **Resource sharing**. Processes may only share resources through techniques such as shared memory or message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than creating a thread, and context switching is about five times slower.
4. **Scalability.** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one processor, regardless how many are available. Multithreading on a multi-CPU machine increases parallelism. We explore this issue further in the following section.

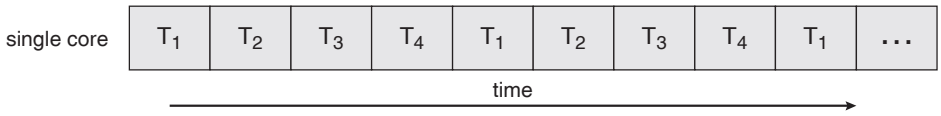


**Figure 4.3** Concurrent execution on a single-core system.

### 4.1.3 Multicore Programming

A recent trend in system design has been to place multiple computing cores on a single chip, where each core appears as a separate processor to the operating system (Section 1.3.2). Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), as the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency means that the threads can run in parallel, as the system can assign a separate thread to each core (Figure 4.4).

The trend towards multicore systems has placed pressure on system designers as well as application programmers to make better use of the multiple computing cores. Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution shown in Figure 4.4. For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded to take advantage of multicore systems. In general, five areas present challenges in programming for multicore systems:

1. **Dividing activities**. This involves examining applications to find areas that can be divided into separate, concurrent tasks and thus can run in parallel on individual cores.
2. **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks; using a separate execution core to run that task may not be worth the cost.
3. **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

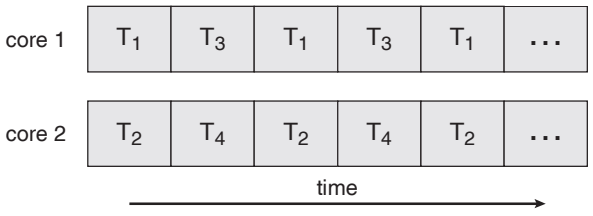


**Figure 4.4** Parallel execution on a multicore system.

4. **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks. In instances where one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency. We examine such strategies in Chapter 6.
5. **Testing and debugging**. When a program is running in parallel on multiple cores, there are many different execution paths. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future.

## 4.2 Multithreading Models

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, Solaris, and Tru64 UNIX (formerly Digital UNIX)—support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship.

### 4.2.1 Many-to-One Model

The many-to-one model (Figure 4.5) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user

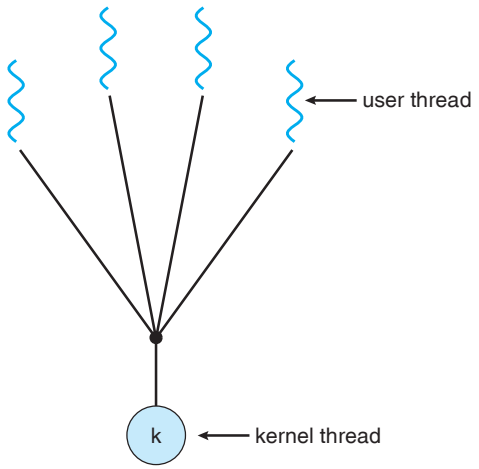


**Figure 4.5** Many-to-one model.

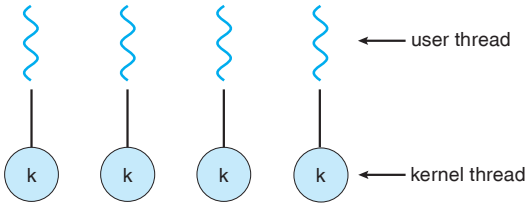


**Figure 4.6** One-to-one model.

space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. **Green threads**—a thread library available for Solaris—uses this model, as does **GNU Portable Threads**.

### 4.2.2 One-to-One Model

The one-to-one model (Figure 4.6) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

### 4.2.3 Many-to-Many Model

The many-to-many model (Figure 4.7) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer to

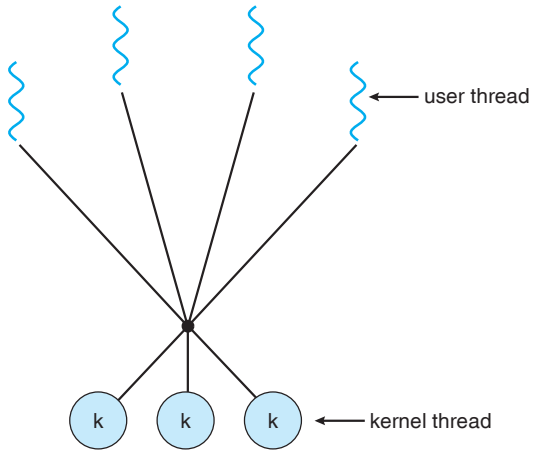


**Figure 4.7** Many-to-many model.

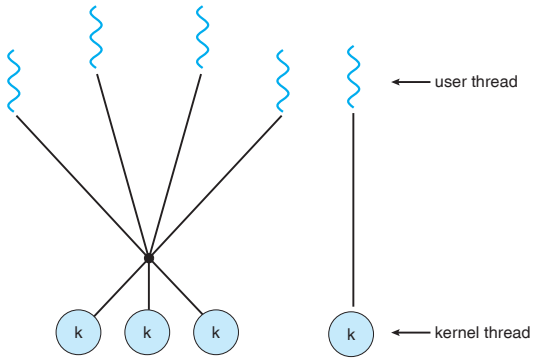


**Figure 4.8** Two-level model.

create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One popular variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation, sometimes referred to as the *two-level model* (Figure 4.8), is supported by operating systems such as IRIX, HP-UX, and Tru64 UNIX. The Solaris operating system supported the two-level model in versions older than Solaris 9. However, beginning with Solaris 9, this system uses the one-to-one model.

## 4.3 Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: (1) POSIX Pthreads, (2) Win32, and (3) Java. Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library. The Win32 thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs. However, because in most instances the JVM is running on top of a host operating system,

the Java thread API is generally implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using the Win32 API; UNIX and Linux systems often use Pthreads.

In the remainder of this section, we describe basic thread creation using these three thread libraries. As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$sum = \sum_{i=0}^{N} i$$

For example, if $N$ were 5, this function would represent the summation of integers from 0 to 5, which is 15. Each of the three programs will be run with the upper bounds of the summation entered on the command line; thus, if the user enters 8, the summation of the integer values from 0 to 8 will be output.

### 4.3.1 Pthreads

**Pthreads** refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a *specification* for thread behavior, not an *implementation*. Operating system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification, including Solaris, Linux, Mac OS X, and Tru64 UNIX. *Shareware* implementations are available in the public domain for the various Windows operating systems as well.

The C program shown in Figure 4.9 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread. In a Pthreads program, separate threads begin execution in a specified function. In Figure 4.9, this is the `runner()` function. When this program begins, a single thread of control begins in `main()`. After some initialization, `main()` creates a second thread that begins control in the `runner()` function. Both threads share the global data `sum`.

Let's look more closely at this program. All Pthreads programs must include the `pthread.h` header file. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The `pthread_attr_t attr` declaration represents the attributes for the thread. We set the attributes in the function call `pthread_attr_init(&attr)`. Because we did not explicitly set any attributes, we use the default attributes provided. (In Chapter 5, we discuss some of the scheduling attributes provided by the Pthreads API.) A separate thread is created with the `pthread_create()` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the `runner()` function. Last, we pass the integer parameter that was provided on the command line, `argv[1]`.

At this point, the program has two threads: the initial (or parent) thread in `main()` and the summation (or child) thread performing the summation

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

**Figure 4.9** Multithreaded C program using the Pthreads API.

operation in the `runner()` function. After creating the summation thread, the parent thread will wait for it to complete by calling the `pthread_join()` function. The summation thread will complete when it calls the function `pthread_exit()`. Once the summation thread has returned, the parent thread will output the value of the shared data `sum`.

### 4.3.2 Win32 Threads

The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways. We illustrate the Win32 thread API in the C program shown in Figure 4.10. Notice that we must include the `windows.h` header file when using the Win32 API.

Just as in the Pthreads version shown in Figure 4.9, data shared by the separate threads—in this case, `Sum`—are declared globally (the `DWORD` data type is an unsigned 32-bit integer). We also define the `Summation()` function that is to be performed in a separate thread. This function is passed a pointer to a `void`, which Win32 defines as LPVOID. The thread performing this function sets the global data `Sum` to the value of the summation from 0 to the parameter passed to `Summation()`.

Threads are created in the Win32 API using the `CreateThread()` function, and—just as in Pthreads—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes (which do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler). Once the summation thread is created, the parent must wait for it to complete before outputting the value of `Sum`, as the value is set by the summation thread. Recall that the Pthread program (Figure 4.9) had the parent thread wait for the summation thread using the `pthread_join()` statement. We perform the equivalent of this in the Win32 API using the `WaitForSingleObject()` function, which causes the creating thread to block until the summation thread has exited. (We cover synchronization objects in more detail in Chapter 6.)

### 4.3.3 Java Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a `main()` method runs as a single thread in the JVM.

There are two techniques for creating threads in a Java program. One approach is to create a new class that is derived from the `Thread` class and to override its `run()` method. An alternative—and more commonly used—technique is to define a class that implements the `Runnable` interface. The `Runnable` interface is defined as follows:

```
public interface Runnable
{
    public abstract void run();
}
```

When a class implements `Runnable`, it must define a `run()` method. The code implementing the `run()` method is what runs as a separate thread.

Figure 4.11 shows the Java version of a multithreaded program that determines the summation of a non-negative integer. The `Summation` class implements the `Runnable` interface. Thread creation is performed by creating

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   // create the thread
   ThreadHandle = CreateThread(
      NULL, // default security attributes
      0, // default stack size
      Summation, // thread function
      &Param, // parameter to thread function
      0, // default creation flags
      &ThreadId); // returns the thread identifier

   if (ThreadHandle != NULL) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

**Figure 4.10** Multithreaded C program using the Win32 API.

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}

public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         // create the object to be shared
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

**Figure 4.11** Java program for the summation of a non-negative integer.

an object instance of the Thread class and passing the constructor a Runnable object.

Creating a Thread object does not specifically create the new thread; rather, it is the start() method that creates the new thread. Calling the start() method for the new object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the run() method, making the thread eligible to be run by the JVM. (Note that we never call the run() method directly. Rather, we call the start() method, and it calls the run() method on our behalf.)

When the summation program runs, two threads are created by the JVM. The first is the parent thread, which starts execution in the main() method. The second thread is created when the start() method on the Thread object is invoked. This child thread begins execution in the run() method of the Summation class. After outputting the value of the summation, this thread terminates when it exits from its run() method.

Sharing of data between threads occurs easily in Win32 and Pthreads, since shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data; if two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads. In the Java program shown in Figure 4.11, the main thread and the summation thread share the object instance of the Sum class. This shared object is referenced through the appropriate getSum() and setSum() methods. (You might wonder why we don't use an Integer object rather than designing a new sum class. The reason is that the Integer class is **immutable**—that is, once its value is set, it cannot change.)

Recall that the parent threads in the Pthreads and Win32 libraries use pthread_join() and WaitForSingleObject() (respectively) to wait for the summation threads to finish before proceeding. The join() method in Java provides similar functionality. (Notice that join() can throw an InterruptedException, which we choose to ignore.)

## 4.4 Threading Issues

In this section, we discuss some of the issues to consider with multithreaded programs.

### 4.4.1 The fork() and exec() System Calls

In Chapter 3, we described how the fork() system call is used to create a separate, duplicate process. The semantics of the fork() and exec() system calls change in a multithreaded program.

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

The exec() system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the exec() system call, the program

**The JVM and the Host Operating System**

The JVM is typically implemented on top of a host operating system (see Figure 2.20). This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that lets Java programs operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying operating system, instead leaving that decision to the particular implementation of the JVM. For example, the Windows operating system uses the one-to-one model; therefore, each Java thread for a JVM running on such a system maps to a kernel thread. On operating systems that use the many-to-many model (such as Tru64 UNIX), a Java thread is mapped according to the many-to-many model. Solaris initially implemented the JVM using the many-to-one model (the green threads library mentioned earlier). Later releases of the JVM were implemented using the many-to-many model. Beginning with Solaris 9, Java threads were mapped using the one-to-one model. In addition, there may be a relationship between the Java thread library and the thread library on the host operating system. For example, implementations of a JVM for Windows might use the Win32 API when creating Java threads; Linux, Solaris, and Mac OS X systems might use the Pthreads API.

specified in the parameter to `exec()` will replace the entire process—including all threads.

Which of the two versions of `fork()` to use depends on the application. If `exec()` is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to `exec()` will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec()` after forking, the separate process should duplicate all threads.

### 4.4.2 Cancellation

**Thread cancellation** is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a Web browser that stops a Web page from loading any further. Often, a Web page is loaded using several threads—each image is loaded in a separate thread. When a user presses the *stop* button on the browser, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.

2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely. Pthreads refers to such points as **cancellation points**.

### 4.4.3 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire. Typically, an asynchronous signal is sent to another process.

A signal may be *handled* by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that is run by the kernel when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends on the type of signal generated. For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process. However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (`<control><C>`, for example)—should be sent to all threads.

Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it. The standard UNIX function for delivering a signal is `kill(pid_t pid, int signal)`, which specifies the process (`pid`) to which a particular signal is to be delivered. POSIX Pthreads provides the `pthread_kill(pthread_t tid, int signal)` function, which allows a signal to be delivered to a specified thread (`tid`).

Although Windows does not explicitly provide support for signals, they can be emulated using **asynchronous procedure calls (APCs)**. The APC facility allows a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process.

### 4.4.4 Thread Pools

In Section 4.1, we mentioned multithreading in a Web server. In this situation, whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems. The first issue concerns the amount of time required to create the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work. The second issue is more troublesome: if we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.

The general idea behind a thread pool is to create a number of threads at process startup and place them into a *pool*, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request for service. Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.

Thread pools offer these benefits:

1. Servicing a request with an existing thread is usually faster than waiting to create a thread.
2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further benefit of having a smaller pool—thereby consuming less memory—when the load on the system is low.

The Win32 API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the `CreateThread()` function, as described in Section 4.3.2. Here, a function that is to run as a separate thread is defined. Such a function may appear as follows:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /**
    * this function runs as a separate thread.
    **/
}
```

A pointer to `PoolFunction()` is passed to one of the functions in the thread pool API, and a thread from the pool executes this function. One such member in the thread pool API is the `QueueUserWorkItem()` function, which is passed three parameters:

- `LPTHREAD_START_ROUTINE Function`—a pointer to the function that is to run as a separate thread
- `PVOID Param`—the parameter passed to `Function`
- `ULONG Flags`—flags indicating how the thread pool is to create and manage execution of the thread

An example of invoking a function is:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

This causes a thread from the thread pool to invoke `PoolFunction()` on behalf of the programmer. In this instance, we pass no parameters to `PoolFunc-`

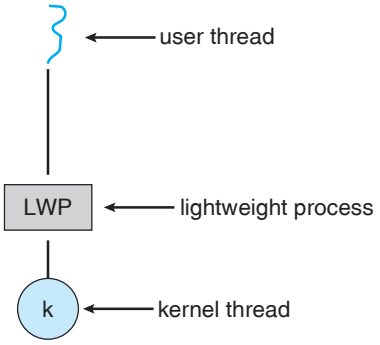


**Figure 4.12** Lightweight process (LWP).

`tion()`. Because we specify `0` as a flag, we provide the thread pool with no special instructions for thread creation.

Other members in the Win32 thread pool API include utilities that invoke functions at periodic intervals or when an asynchronous I/O request completes. The `java.util.concurrent` package in Java 1.5 provides a thread pool utility as well.

### 4.4.5 Thread-Specific Data

Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-specific data**. For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data. Most thread libraries—including Win32 and Pthreads—provide some form of support for thread-specific data. Java provides support as well.

### 4.4.6 Scheduler Activations

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models discussed in Section 4.2.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a lightweight process, or LWP—is shown in Figure 4.12. To the user-thread library, the LWP appears to be a *virtual processor* on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only

one thread can run at once, so one LWP is sufficient. An application that is I/O-intensive may require multiple LWPs to execute, however. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor. One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

## 4.5 Operating-System Examples

In this section, we explore how threads are implemented in Windows XP and Linux systems.

### 4.5.1 Windows Threads

Windows implements the Win32 API as its primary API. A Windows application runs as a separate process, and each process may contain one or more threads. The Win32 API for creating threads is covered in Section 4.3.2. Windows uses the one-to-one mapping described in Section 4.2.2, where each user-level thread maps to an associated kernel thread. However, Windows also provides support for a **fiber** library, which provides the functionality of the many-to-many model (Section 4.2.3). By using the thread library, any thread belonging to a process can access the address space of the process.

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor

- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
- A private storage area used by various run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the **context** of the thread. The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB—thread environment block

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them. The TEB is a user-space data structure that

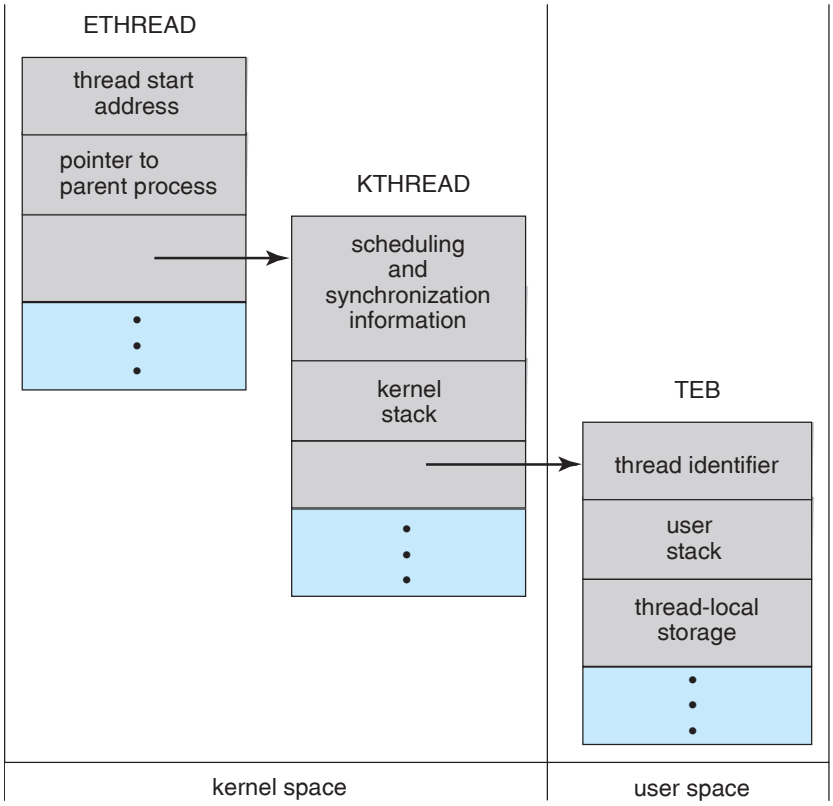


**Figure 4.13** Data structures of a Windows thread.

is accessed when the thread is running in user mode. Among other fields, the TEB contains the thread identifier, a user-mode stack, and an array for thread-specific data (which Windows terms **thread-local storage**). The structure of a Windows thread is illustrated in Figure 4.13.

### 4.5.2 Linux Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process, as described in Chapter 3. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux generally uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program.

When `clone()` is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed below:

| flag | meaning |
|---|---|
| `CLONE_FS` | File-system information is shared. |
| `CLONE_VM` | The same memory space is shared. |
| `CLONE_SIGHAND` | Signal handlers are shared. |
| `CLONE_FILES` | The set of open files is shared. |

For example, if `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task. However, if none of these flags is set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to that provided by the `fork()` system call.

The varying level of sharing is possible because of the way a task is represented in the Linux kernel. A unique kernel data structure (specifically, `struct task_struct`) exists for each task in the system. This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored—for example, data structures that represent the list of open files, signal-handling information, and virtual memory. When `fork()` is invoked, a new task is created, along with a *copy* of all the associated data structures of the parent process. A new task is also created when the `clone()` system call is made. However, rather than copying all data structures, the new task *points* to the data structures of the parent task, depending on the set of flags passed to `clone()`.

Several distributions of the Linux kernel now include the NPTL thread library. NPTL (which stands for Native POSIX Thread Library) provides a POSIX-compliant thread model for Linux systems along with several other features,

such as better support for SMP systems, as well as taking advantage of NUMA support. In addition, the start-up cost for creating a thread is lower with NPTL than with traditional Linux threads. Finally, with NPTL, the system has the potential to support hundreds of thousands of threads. Such support becomes more important with the growth of multicore and other SMP systems.

## 4.6 Summary

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and scalability issues such as more efficient use of multiple cores.

User-level threads are threads that are visible to the programmer and are unknown to the kernel. The operating-system kernel supports and manages kernel-level threads. In general, user-level threads are faster to create and manage than are kernel threads, as no intervention from the kernel is required. Three different types of models relate user and kernel threads: The many-to-one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.

Most modern operating systems provide kernel support for threads; among these are Windows 98, NT, 2000, and XP, as well as Solaris and Linux.

Thread libraries provide the application programmer with an API for creating and managing threads. Three primary thread libraries are in common use: POSIX Pthreads, Win32 threads for Windows systems, and Java threads.

Multithreaded programs introduce many challenges for the programmer, including the semantics of the `fork()` and `exec()` system calls. Other issues include thread cancellation, signal handling, and thread-specific data.

## Practice Exercises

**4.1** Provide two programming examples in which multithreading provides better performance than a single-threaded solution.

**4.2** What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

**4.3** Describe the actions taken by a kernel to context-switch between kernel-level threads.

**4.4** What resources are used when a thread is created? How do they differ from those used when a process is created?

**4.5** Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

**4.6** A Pthread program that performs the summation function was provided in Section 4.3.1. Rewrite this program in Java.

## Exercises

**4.7** Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution.

**4.8** Describe the actions taken by a thread library to context-switch between user-level threads.

**4.9** Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

**4.10** Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

**4.11** Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

**4.12** As described in Section 4.5.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. However, many operating systems—such as Windows and Solaris—treat processes and threads differently. Typically, such systems use a notation wherein the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

**4.13** The program shown in Figure 4.14 uses the Pthreads API. What would be the output from the program at `LINE C` and `LINE P`?

**4.14** Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processors in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processors.

b. The number of kernel threads allocated to the program is equal to the number of processors.

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
int pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

**Figure 4.14** C program for Exercise 4.13.

c. The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user-level threads.

**4.15** Write a multithreaded Java, Pthreads, or Win32 program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

**4.16** Modify the socket-based date server (Figure 3.19) in Chapter 3 so that the server services each client request in a separate thread.

**4.17** The Fibonacci sequence is the series of numbers $0, 1, 1, 2, 3, 5, 8, \ldots$. Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a multithreaded program that generates the Fibonacci sequence using either the Java, Pthreads, or Win32 thread library. This program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish, using the techniques described in Section 4.3.

**4.18** Exercise 3.17 in Chapter 3 involves designing an echo server using the Java threading API. However, this server is single-threaded, meaning that the server cannot respond to concurrent echo clients until the current client exits. Modify the solution to Exercise 3.17 so that the echo server services each client in a separate request.

## Programming Projects

The set of projects below deal with two distinct topics—naming service and matrix muliplication.

### Project 1: Naming Service Project

A naming service such as DNS (for domain name system) can be used to resolve IP names to IP addresses. For example, when someone accesses the host `www.westminstercollege.edu`, a naming service is used to determine the IP address that is mapped to the IP name `www.westminstercollege.edu`. This assignment consists of writing a multithreaded naming service in Java using sockets (see Section 3.6.1).

The `java.net` API provides the following mechanism for resolving IP names:

```
InetAddress hostAddress =
    InetAddress.getByName("www.westminstercollege.edu");
String IPaddress = hostAddress.getHostAddress();
```

where `getByName()` throws an `UnknownHostException` if it is unable to resolve the host name.

#### The Server

The server will listen to port 6052 waiting for client connections. When a client connection is made, the server will service the connection in a separate thread and will resume listening for additional client connections. Once a client makes a connection to the server, the client will write the IP name it wishes the server to resolve—such as `www.westminstercollege.edu`—to the socket. The server thread will read this IP name from the socket and either resolve its IP address or, if it cannot locate the host address, catch an

UnknownHostException. The server will write the IP address back to the client or, in the case of an UnknownHostException, will write the message "Unable to resolve host <host name>." Once the server has written to the client, it will close its socket connection.

### The Client

Initially, write just the server application and connect to it via telnet. For example, assuming the server is running on the localhost, a telnet session would appear as follows. (Client responses appear in blue.)

```
telnet localhost 6052
Connected to localhost.
Escape character is '^]'.
www.westminstercollege.edu
146.86.1.17
Connection closed by foreign host.
```

By initially having telnet act as a client, you can more easily debug any problems you may have with your server. Once you are convinced your server is working properly, you can write a client application. The client will be passed the IP name that is to be resolved as a parameter. The client will open a socket connection to the server and then write the IP name that is to be resolved. It will then read the response sent back by the server. As an example, if the client is named NSClient, it is invoked as follows:

```
java NSClient www.westminstercollege.edu
```

and the server will respond with the corresponding IP address or "unknown host" message. Once the client has output the IP address, it will close its socket connection.

## Project 2: Matrix Multiplication Project

Given two matrices, $A$ and $B$, where matrix $A$ contains $M$ rows and $K$ columns and matrix $B$ contains $K$ rows and $N$ columns, the **matrix product** of $A$ and $B$ is matrix $C$, where $C$ contains $M$ rows and $N$ columns. The entry in matrix $C$ for row $i$, column $j$ ($C_{i,j}$) is the sum of the products of the elements for row $i$ in matrix $A$ and column $j$ in matrix $B$. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

For example, if $A$ is a 3-by-2 matrix and $B$ is a 2-by-3 matrix, element $C_{3,1}$ is the sum of $A_{3,1} \times B_{1,1}$ and $A_{3,2} \times B_{2,1}$.

For this project, calculate each element $C_{i,j}$ in a separate *worker* thread. This will involve creating $M \times N$ worker threads. The main—or parent—thread will initialize the matrices $A$ and $B$ and allocate sufficient memory for matrix $C$, which will hold the product of matrices $A$ and $B$. These matrices will be declared as global data so that each worker thread has access to $A$, $B$, and $C$.

Matrices $A$ and $B$ can be initialized statically, as shown below:

```
#define M 3
#define K 2
#define N 3

int A [M][K] = { {1,4}, {2,5}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

Alternatively, they can be populated by reading in values from a file.

**Passing Parameters to Each Thread**

The parent thread will create $M \times N$ worker threads, passing each worker the values of row $i$ and column $j$ that it is to use in calculating the matrix product. This requires passing two parameters to each thread. The easiest approach with Pthreads and Win32 is to create a data structure using a `struct`. The members of this structure are $i$ and $j$, and the structure appears as follows:

```
/* structure for passing data to threads */
struct v
{
   int i; /* row */
   int j; /* column */
};
```

Both the Pthreads and Win32 programs will create the worker threads using a strategy similar to that shown below:

```
/* We have to create M * N worker threads */
for (i = 0; i < M, i++)
   for (j = 0; j < N; j++ ) {
      struct v *data = (struct v *) malloc(sizeof(struct v));
      data->i = i;
      data->j = j;
      /* Now create the thread passing it data as a parameter */
   }
}
```

The `data` pointer will be passed to either the `pthread_create()` (Pthreads) function or the `CreateThread()` (Win32) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

Sharing of data between Java threads is different from sharing between threads in Pthreads or Win32. One approach is for the main thread to create and initialize the matrices $A$, $B$, and $C$. This main thread will then create the worker threads, passing the three matrices—along with row $i$ and column $j$—to the constructor for each worker. Thus, the outline of a worker thread appears in Figure 4.15.

```
public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

   public WorkerThread(int row, int col, int[][] A,
    int[][] B, int[][] C) {
      this.row = row;
      this.col = col;
      this.A = A;
      this.B = B;
      this.C = C;
   }
   public void run() {
      /* calculate the matrix product in C[row] [col] */
   }
}
```

**Figure 4.15** Worker thread in Java.

### Waiting for Threads to Complete

Once all worker threads have completed, the main thread will output the product contained in matrix $C$. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait for other threads to finish. Section 4.3 describes how to wait for a child thread to complete using the Win32, Pthreads, and Java thread libraries. Win32 provides the `WaitForSingleObject()` function, whereas Pthreads and Java use `pthread_join()` and `join()`, respectively. However, in these programming examples, the parent thread waits for a single child thread to finish; completing this exercise will require waiting for multiple threads.

In Section 4.3.2, we describe the `WaitForSingleObject()` function, which is used to wait for a single thread to finish. However, the Win32 API also provides the `WaitForMultipleObjects()` function, which is used when

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

**Figure 4.16** Pthread code for joining ten threads.

```
final static int NUM_THREADS = 10;

/* an array of threads to be joined upon */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
   try {
      workers[i].join();
    } catch (InterruptedException ie) { }
}
```

**Figure 4.17** Java code for joining ten threads.

waiting for multiple threads to complete. `WaitForMultipleObjects()` is passed four parameters:

1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicating if all objects have been signaled
4. A timeout duration (or `INFINITE`)

For example, if `THandles` is an array of thread `HANDLE` objects of size `N`, the parent thread can wait for all its child threads to complete with the statement

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

A simple strategy for waiting on several threads using the Pthreads `pthread_join()` or Java's `join()` is to enclose the join operation within a simple `for` loop. For example, you could join on ten threads using the Pthread code depicted in Figure 4.16. The equivalent code using Java threads is shown in Figure 4.17.

## Bibliographical Notes

Threads have had a long evolution, starting as "cheap concurrency" in programming languages and moving to "lightweight processes", with early examples that included the Thoth system (Cheriton et al. [1979]) and the Pilot system (Redell et al. [1980]). Binding [1985] described moving threads into the UNIX kernel. Mach (Accetta et al. [1986], Tevanian et al. [1987]) and V (Cheriton [1988]) made extensive use of threads, and eventually almost all major operating systems have implemented them in some form or another.

Thread-performance issues were discussed by Anderson et al. [1989], who continued their work in Anderson et al. [1991] by evaluating the performance of user-level threads with kernel support. Bershad et al. [1990] describe combining threads with RPC. Engelschall [2000] discusses a technique for supporting user-level threads. An analysis of an optimal thread-pool size can be found in Ling et al. [2000]. Scheduler activations were first presented in Anderson et al. [1991], and Williams [2002] discusses scheduler activations in

the NetBSD system. Other mechanisms by which the user-level thread library and the kernel cooperate with each other are discussed in Marsh et al. [1991], Govindan and Anderson [1991], Draves et al. [1991], and Black [1990]. Zabatta and Young [1998] compare Windows NT and Solaris threads on a symmetric multiprocessor. Pinilla and Gill [2003] compare Java thread performance on Linux, Windows, and Solaris.

Vahalia [1996] covers threading in several versions of UNIX. McDougall and Mauro [2007] describe recent developments in threading the Solaris kernel. Russinovich and Solomon [2009] discuss threading in the Windows operating system family. Bovet and Cesati [2006] and Love [2004] explain how Linux handles threading and Singh [2007] covers threads in Mac OS X.

Information on Pthreads programming is given in Lewis and Berg [1998] and Butenhof [1997]. Oaks and Wong [1999], Lewis and Berg [2000], and Holub [2000] discuss multithreading in Java. Goetz et al. [2006] present a detailed discussion of concurrent programming in Java. Beveridge and Wiener [1997] and Cohen and Woodring [1997] describe multithreading using Win32.