

# Threads



The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control. In this chapter, we introduce many concepts associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Windows, and Java thread libraries. We look at a number of 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

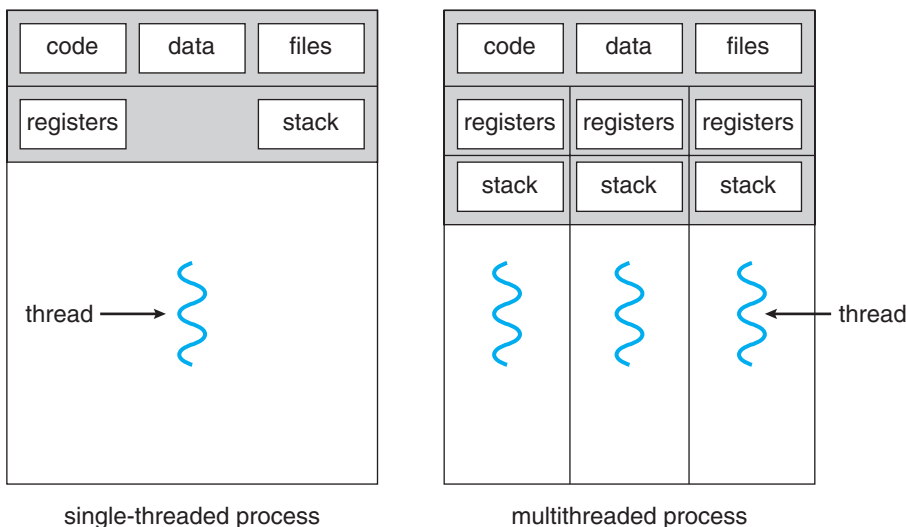
- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries.
- To explore several strategies that provide implicit threading.
- To examine issues related to multithreaded programming.
- To cover operating system support for threads in Windows and Linux.

## 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

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several



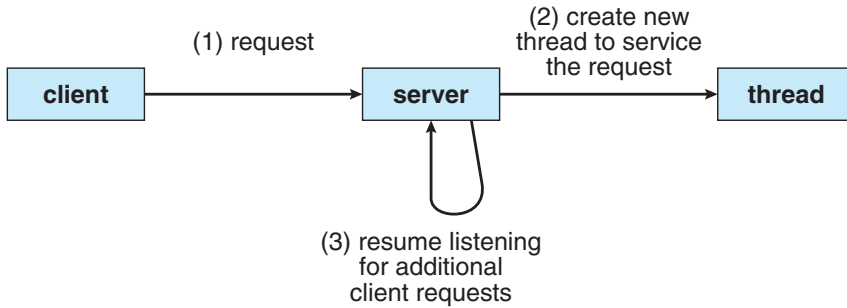
**Figure 4.1** Single-threaded and multithreaded processes.

threads of control. A web browser might have one thread display images or 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. Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

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 creates 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



**Figure 4.2** Multithreaded server architecture.

services the message using a separate thread. This allows the server to service 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 as managing devices, managing memory, or interrupt handling. For example, Solaris has 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. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.
2. **Resource sharing.** Processes can only share resources through techniques such as shared memory and 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 significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times

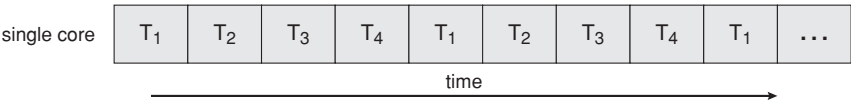


Figure 4.3 Concurrent execution on a single-core system.

slower than is creating a thread, and context switching is about five times slower.

- 4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

4.2 Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system (Section 1.3.2). Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing 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), because 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, because the system can assign a separate thread to each core (Figure 4.4).

Notice the distinction between **parallelism** and **concurrency** in this discussion. **A system is parallel if it can perform more than one task simultaneously.** In contrast, **a concurrent system supports more than one task by allowing all the tasks to make progress.** Thus, it is possible to have concurrency without parallelism. Before the advent of SMP and multicore architectures, most computer systems had only a single processor. CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in

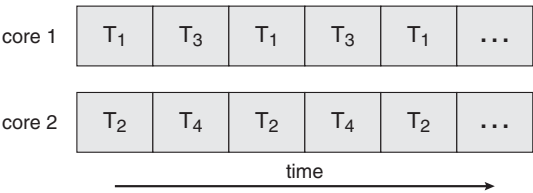


Figure 4.4 Parallel execution on a multicore system.

### AMDAHL'S LAW

Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If  $S$  is the portion of the application that must be performed serially on a system with  $N$  processing cores, the formula appears as follows:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add two additional cores (for a total of four), the speedup is 2.28 times.

One interesting fact about Amdahl's Law is that as  $N$  approaches infinity, the speedup converges to  $1/S$ . For example, if 40 percent of an application is performed serially, the maximum speedup is 2.5 times, regardless of the number of processing cores we add. This is the fundamental principle behind Amdahl's Law: the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

Some argue that Amdahl's Law does not take into account the hardware performance enhancements used in the design of contemporary multicore systems. Such arguments suggest Amdahl's Law may cease to be applicable as the number of processing cores continues to increase on modern computer systems.

the system, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

As systems have grown from tens of threads to thousands of threads, CPU designers have improved system performance by adding hardware to improve thread performance. Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight threads per core. This support means that multiple threads can be loaded into the core for fast switching. Multicore computers will no doubt continue to increase in core counts and hardware thread support.

#### 4.2.1 Programming Challenges

The trend towards multicore systems continues to place pressure on system designers and 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.

In general, five areas present challenges in programming for multicore systems:

1. **Identifying tasks.** This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another 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.
4. **Data dependency.** The data accessed by the tasks must be examined for dependencies between two or more tasks. When 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 5.
5. **Testing and debugging.** When a program is running in parallel on multiple cores, many different execution paths are possible. 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. (Similarly, many computer science educators believe that software development must be taught with increased emphasis on parallel programming.)

#### 4.2.2 Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Consider, for example, summing the contents of an array of size  $N$ . On a single-core system, one thread would simply sum the elements  $[0] \dots [N - 1]$ . On a dual-core system, however, thread  $A$ , running on core 0, could sum the elements  $[0] \dots [N/2 - 1]$  while thread  $B$ , running on core 1, could sum the elements  $[N/2] \dots [N - 1]$ . The two threads would be running in parallel on separate computing cores.

**Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data. Consider again our example above. In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

Fundamentally, then, data parallelism involves the distribution of data across multiple cores and task parallelism on the distribution of tasks across multiple cores. In practice, however, few applications strictly follow either data or task parallelism. In most instances, applications use a hybrid of these two strategies.

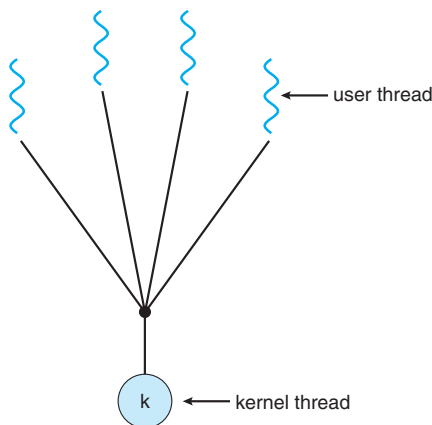
## 4.3 Multithreading Models

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, and Solaris—support kernel threads.

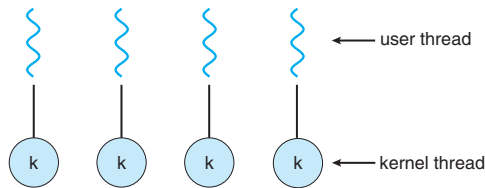
Ultimately, a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

### 4.3.1 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 space, so it is efficient (we discuss thread libraries in Section 4.4). However, 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 multicore systems. **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model. However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores.



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



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

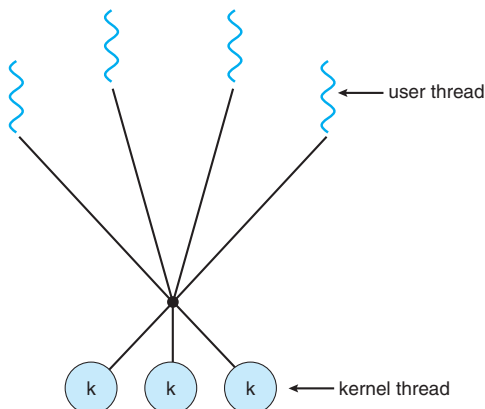
#### 4.3.2 One-to-One Model

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.

#### 4.3.3 Many-to-Many Model

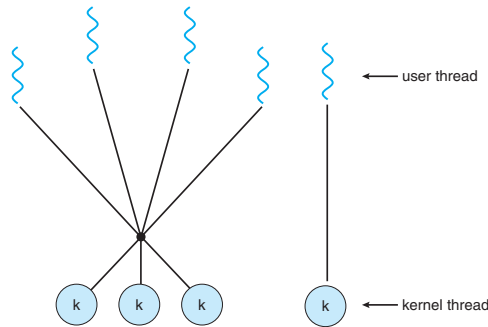
The many-to-many model (Figure 4.7) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than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

Let's consider the effect of this design on concurrency. Whereas the many-to-one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time. The one-to-one model allows 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



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





**Figure 4.8** Two-level model.

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 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 is sometimes referred to as the **two-level model** (Figure 4.8). 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.

## 4.4 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: POSIX Pthreads, Windows, and Java. Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library. The Windows 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 Windows API; UNIX and Linux systems often use Pthreads.

### THE JVM AND THE HOST OPERATING SYSTEM

The JVM is typically implemented on top of a host operating system (see Figure 16.10). This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to 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 XP 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 the Windows family of operating systems might use the Windows API when creating Java threads; Linux, Solaris, and Mac OS X systems might use the Pthreads API.

threading from application developers to compilers and run-time libraries. This strategy, termed **implicit threading**, is a popular trend today. In this section, we explore three alternative approaches for designing multithreaded programs that can take advantage of multicore processors through implicit threading.

#### 4.5.1 Thread Pools

In Section 4.1, we described a multithreaded 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, together with the fact that the 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 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.
3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

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. We discuss one such architecture, Apple’s Grand Central Dispatch, later in this section.

The Windows API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the `Thread.Create()` function, as described in Section 4.4.2. 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 the following:

```
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-`

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

Other members in the Windows 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 the Java API provides a thread-pool utility as well.

### 4.5.2 OpenMP

OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel. The following C program illustrates a compiler directive above the parallel region containing the `printf()` statement:

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

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```

When OpenMP encounters the directive

```
#pragma omp parallel
```

it creates as many threads as there are processing cores in the system. Thus, for a dual-core system, two threads are created, for a quad-core system, four are created; and so forth. All the threads then simultaneously execute the parallel region. As each thread exits the parallel region, it is terminated.

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops. For example, assume we have two arrays `a` and `b` of size `N`. We wish to sum their contents and place the results in array `c`. We can have this task run in parallel by using the following code segment, which contains the compiler directive for parallelizing `for` loops:

The standard UNIX function for delivering a signal is

```
kill(pid_t pid, int signal)
```

This function specifies the process (`pid`) to which a particular signal (`signal`) is to be delivered. 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. POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (`tid`):

```
pthread_kill(pthread_t tid, int signal)
```

Although Windows does not explicitly provide support for signals, it allows us to emulate them using **asynchronous procedure calls (APCs)**. The APC facility enables 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.

#### 4.6.3 Thread Cancellation

**Thread cancellation** involves 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 loads 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.

In Pthreads, thread cancellation is initiated using the `pthread_cancel()` function. The identifier of the target thread is passed as a parameter to the function. The following code illustrates creating—and then canceling—a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Invoking `pthread_cancel()` indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request. Pthreads supports three cancellation modes. Each mode is defined as a state and a type, as illustrated in the table below. A thread may set its cancellation state and type using an API.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

As the table illustrates, Pthreads allows threads to disable or enable cancellation. Obviously, a thread cannot be canceled if cancellation is disabled. However, cancellation requests remain pending, so the thread can later enable cancellation and respond to the request.

The default cancellation type is deferred cancellation. Here, cancellation occurs only when a thread reaches a cancellation point. One technique for establishing a cancellation point is to invoke the `pthread_testcancel()` function. If a cancellation request is found to be pending, a function known as a **cleanup handler** is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.

The following code illustrates how a thread may respond to a cancellation request using deferred cancellation:

```
while (1) {
    /* do some work for awhile */
    /* . . . */

    /* check if there is a cancellation request */
    pthread_testcancel();
}
```