

```

static VOID MergeArrays (LPRECORD, DWORD);

DWORD WINAPI SortThread (PTHREADARG pThArg)
{
    DWORD groupSize = 2, myNumber, twoToI = 1;
        /* twoToI = 2^i, where i is the merge step number. */
    DWORD_PTR numbersInGroup;
    LPRECORD first;

    myNumber = pThArg->iTh;
    first = pThArg->lowRecord;
    numbersInGroup = pThArg->highRecord - first;

    /* Sort this portion of the array. */
    qsort (first, numbersInGroup, RECSIZE, KeyCompare);

    /* Either exit the thread or wait for the adjoining thread. */
    while ((myNumber % groupSize) == 0 && numbersInGroup < nRec) {
        /* Merge with the adjacent sorted array. */
        WaitForSingleObject (pThreadHandle[myNumber + twoToI],
                            INFINITE);
        MergeArrays (first, numbersInGroup);
        numbersInGroup *= 2;
        groupSize *= 2;
        twoToI *= 2;
    }
    return 0;
}

static VOID MergeArrays (LPRECORD p1, DWORD nRecs)
{
    /* Merge adjacent arrays, with nRecs records. p1 is the first */
    DWORD iRec = 0, i1 = 0, i2 = 0;
    LPRECORD pDest, p1Hold, pDestHold, p2 = p1 + nRecs;

    pDest = pDestHold = malloc (2 * nRecs * RECSIZE);
    p1Hold = p1;

    while (i1 < nRecs && i2 < nRecs) {
        if (KeyCompare ((LPCTSTR)p1, (LPCTSTR)p2) <= 0) {
            memcpy (pDest, p1, RECSIZE);
            i1++; p1++; pDest++;
        }
        else {
            memcpy (pDest, p2, RECSIZE);
            i2++; p2++; pDest++;
        }
    }
    if (i1 >= nRecs)
        memcpy (pDest, p2, RECSIZE * (nRecs - i2));
}

```

```

else
    memcpy (pDest, p1, RECSIZE * (nRecs - i1));

    memcpy (p1Hold, pDestHold, 2 * nRecs * RECSIZE);
    free (pDestHold);
    return;
}

```

```

C:\WSP4_Examples\run8>randfile 4 small.txt

C:\WSP4_Examples\run8>sortMT 1 small.txt
0a16aace. Record Number: 00000001.abcdefghijklmnopqrstuvwxyz x
9d05adef. Record Number: 00000002.abcdefghijklmnopqrstuvwxyz x
b91223c9. Record Number: 00000000.abcdefghijklmnopqrstuvwxyz x
cc4410fc. Record Number: 00000003.abcdefghijklmnopqrstuvwxyz x

C:\WSP4_Examples\run8>randfile 1000000 large.txt

C:\WSP4_Examples\run8>timep sortMT -n 1 large.txt
Real Time: 00:00:03:935
User Time: 00:00:03:790
Sys Time: 00:00:00:062

C:\WSP4_Examples\run8>timep sortMT -n 2 large.txt
Real Time: 00:00:00:955
User Time: 00:00:01:606
Sys Time: 00:00:00:093

C:\WSP4_Examples\run8>timep sortMT -n 4 large.txt
Real Time: 00:00:00:661
User Time: 00:00:01:669
Sys Time: 00:00:00:124

C:\WSP4_Examples\run8>timep sortMT -n 8 large.txt
Real Time: 00:00:00:700
User Time: 00:00:01:747
Sys Time: 00:00:00:280

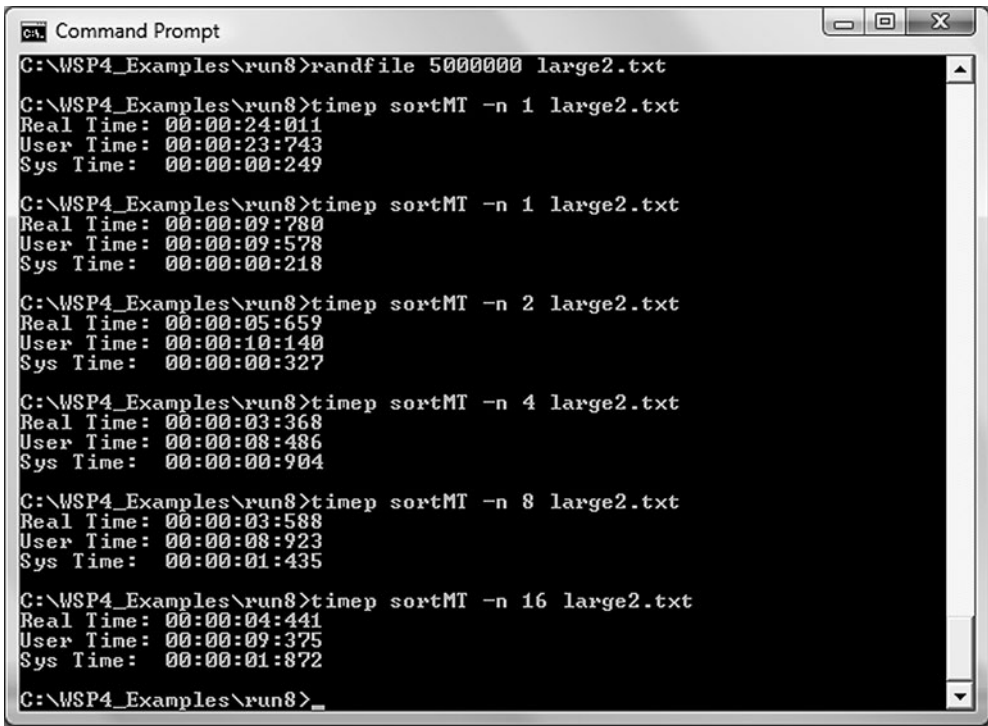
C:\WSP4_Examples\run8>timep sortMT -n 1 large.txt
Real Time: 00:00:02:252
User Time: 00:00:02:215
Sys Time: 00:00:00:000

C:\WSP4_Examples\run8>_

```

Run 7-2a sortMT: Sorting with Multiple Threads

Run 7-2a shows sorting of large and small files, with 1, 2, 4, and 8 threads for the large file. The test computer has four processors, and four threads give the best results. Also note that the first single-thread run is slower than the second; this may be explained by the fact that the file is cached in memory during the second run. sortFL (Program 5-4) was the best previous result, 3.123 seconds.



```
CA: Command Prompt
C:\WSP4_Examples\run8>randfile 5000000 large2.txt
C:\WSP4_Examples\run8>tinep sortMT -n 1 large2.txt
Real Time: 00:00:24:011
User Time: 00:00:23:743
Sys Time: 00:00:00:249
C:\WSP4_Examples\run8>tinep sortMT -n 1 large2.txt
Real Time: 00:00:09:780
User Time: 00:00:09:578
Sys Time: 00:00:00:218
C:\WSP4_Examples\run8>tinep sortMT -n 2 large2.txt
Real Time: 00:00:05:659
User Time: 00:00:10:140
Sys Time: 00:00:00:327
C:\WSP4_Examples\run8>tinep sortMT -n 4 large2.txt
Real Time: 00:00:03:368
User Time: 00:00:08:486
Sys Time: 00:00:00:904
C:\WSP4_Examples\run8>tinep sortMT -n 8 large2.txt
Real Time: 00:00:03:588
User Time: 00:00:08:923
Sys Time: 00:00:01:435
C:\WSP4_Examples\run8>tinep sortMT -n 16 large2.txt
Real Time: 00:00:04:441
User Time: 00:00:09:375
Sys Time: 00:00:01:872
C:\WSP4_Examples\run8>_
```

Run 7-2b sortMT: Sorting with Multiple Threads and a Larger File

An additional screenshot, Run 7-2b, uses a 5,000,000 record (320MB) file so that the time improvements are more significant.

Performance

Multiprocessor systems give good results when the number of threads is the same as the number of processors. Performance improves with more threads but not linearly because of the merging. Additional threads beyond the processor count slow the program.

Divide and conquer is more than just a strategy for algorithm design; it can also be the key to exploiting multiprocessors. The single-processor results can vary. On a computer with limited memory (that is, insufficient physical memory to hold the entire file), using multiple threads might increase the sort time because the threads contend for available physical memory. On the other hand, multiple threads can improve performance with a single processor when there is sufficient memory. The results are also heavily dependent on the initial data arrangement.

Introduction to Program Parallelism

Programs 7–1 and 7–2 share some common properties that permit “parallelization” so that subtasks can execute concurrently, or “in parallel” on separate processors. Parallelization is the key to future performance improvement, since we can no longer depend on increased CPU clock rates and since multicore and multiprocessor systems are increasingly common.

Chapter 10 discusses these technology trends and parallelism in more detail and relates these trends to the thread pools available in NT6 (Windows 7, Vista, and Server 2008). However, `sortMT`, `wcMT`, and `grepMT` have already illustrated the potential performance benefits from parallelism. The properties that enabled parallelism include the following:

- There are separate worker subtasks that can run independently, without any interaction between them. For example, `grepMT` can process each file independently, and `sortMT` can sort subsets of the entire array.
- As subtasks complete, a master program can combine, or “reduce,” the results of several subtasks into a single result. Thus, `sortMT` merges sorted arrays to form larger sorted arrays. `grepMT` and `wcMT` simply display the results from the individual files, in order.
- The programs are “lock-free” and do not need to use mutual exclusion locks, such as the mutexes described next in Chapter 8. The only synchronization required is for the boss thread to wait for the worker threads to complete.
- The worker subtasks run as individual threads, potentially running on separate processors.
- Program performance scales automatically, up to some limit, as you run on systems with more processors; the programs themselves do not, in general, determine the processor count on the host computer. Instead, the Windows kernel assigns worker subtasks to available processors.
- If you “serialize” the program by replacing the thread creation calls with direct function calls and remove the wait calls, you should get precisely the same results as the parallel program.² The serialized program is, moreover, much easier to debug.

² This statement fails, or is only approximately true, if operation order and associativity are important. For example, if you sum floating-point numbers, the order is important. In these cases, the multi-threaded results will also vary from run to run, and the serial execution is one of many possible multi-threaded execution sequences.

- The maximum performance improvement is limited by the program’s “parallelism,” thread management overhead, and computations that cannot be parallelized. The maximum parallelism for `sortMT` is determined by the command line parameter specifying the number of threads, although the merging steps do not use all the threads. `grepMT` parallelism cannot be larger than the number of files on the command line. Computations that cannot be parallelized include initialization and reducing worker results.

Be aware, however, that these two examples are relatively simple and “coarse grained.” The subtasks are easy to identify and run for a relatively long time period, although the subtasks will require different amounts of time, depending primarily on the file sizes. In general, correct program parallelization that improves performance significantly can be challenging.

Thread Local Storage

Threads may need to allocate and manage their own storage independently of and protected from other threads in the same process. One technique is to have the creating thread call `CreateThread` (or `_beginthreadex`) with `lpvThreadParm` pointing to a data structure that is unique for each thread. The thread can then allocate additional data structures and access them through `lpvThreadParm`. Program 7–1 used this technique.

Windows also provides TLS, which gives each thread its own array of pointers. Figure 7–3 shows this TLS arrangement.

Initially, no TLS indexes (rows) are allocated, but new rows can be allocated and deallocated at any time, with at least `TLS_MINIMUM_AVAILABLE` (64) indexes

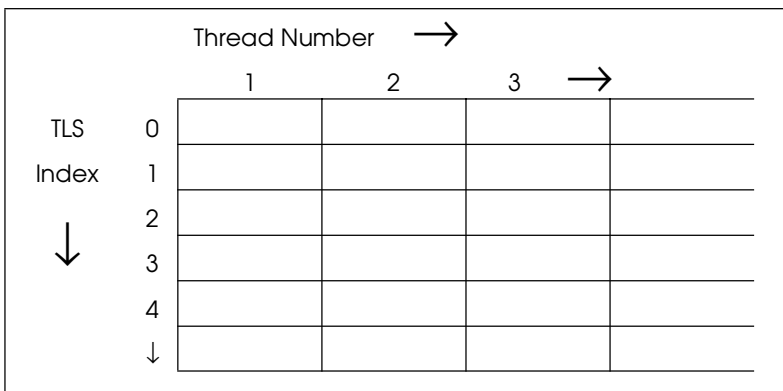


Figure 7–3 Thread Local Storage within a Process

for any process. The number of columns can change as new threads are created and old ones terminate.

The first issue is TLS index management. The primary thread is a logical place to do this, but any thread can manage thread indexes.

`TlsAlloc` returns the allocated index (≥ 0), with -1 (`0xFFFFFFFF`) if no index is available.

```
DWORD TlsAlloc (VOID)

BOOL TlsFree (DWORD dwIndex)
```

An individual thread can get and set its values (void pointers) from its slot using a TLS index.

The programmer must ensure that the TLS index parameter is valid—that is, that it has been allocated with `TlsAlloc` and has not been freed.

```
LPVOID TlsGetValue (DWORD dwTlsIndex)

BOOL TlsSetValue (DWORD dwTlsIndex,
    LPVOID lpTlsValue)
```

TLS provides a convenient mechanism for storage that is global within a thread but unavailable to other threads. Normal global storage is shared by all threads. Although no thread can access another thread's TLS, any thread can call `TlsFree` and destroy an index for all threads, so use `TlsFree` carefully. TLS is frequently used by DLLs as a replacement for global storage in a library; each thread, in effect, has its own global storage. TLS also provides a convenient way for a calling program to communicate with a DLL function, and this is the most common TLS use. An example in Chapter 12 (Program 12–5) exploits TLS to build a thread-safe DLL; DLL thread and process attach/detach calls (Chapter 5) are another important element in the solution.

Process and Thread Priority and Scheduling

The Windows kernel always runs the highest-priority thread that is ready for execution. A thread is not ready if it is waiting, suspended, or blocked for some reason.

Threads receive priority relative to their process priority classes. Process priority classes are set initially by `CreateProcess` (Chapter 6), and each has a *base priority*, with values including:

- `IDLE_PRIORITY_CLASS`, for threads that will run only when the system is idle.
- `NORMAL_PRIORITY_CLASS`, indicating no special scheduling requirements.
- `HIGH_PRIORITY_CLASS`, indicating time-critical tasks that should be executed immediately.
- `REALTIME_PRIORITY_CLASS`, the highest possible priority.

The two extreme classes are rarely used, and the normal class can be used normally, as the name suggests. Windows is not a real-time OS, and using `REALTIME_PRIORITY_CLASS` can prevent other essential threads from running.

Set and get the priority class with:

```
BOOL SetPriorityClass (HANDLE hProcess,
    DWORD dwPriority)

DWORD GetPriorityClass (HANDLE hProcess)
```

You can use the values listed above as well as:

- Two additional priority classes, `ABOVE_NORMAL_PRIORITY_CLASS` (which is below `HIGH_PRIORITY_CLASS`) and `BELOW_NORMAL_PRIORITY_CLASS` (which is above `IDLE_PRIORITY_CLASS`).
- `PROCESS_MODE_BACKGROUND_BEGIN`, which lowers the priority of the process and its threads for background work without affecting the responsiveness of foreground³ processes and threads. The handle must represent the calling process; a process cannot put another into background mode. You need NT6 (Windows Vista or later) to use this mode.
- `PROCESS_MODE_BACKGROUND_END`, which restores the process priority to the value before it was set with `PROCESS_MODE_BACKGROUND_BEGIN`.

³ Foreground threads and processes (“tasks”) are generally those that need to respond quickly, such as a thread that interacts directly with the user. Background tasks do not need to respond quickly; examples include file processing or time-consuming computations.

A process can change or determine its own priority or that of another process, security permitting.

Thread priorities are either absolute or are set relative to the process base priority. At thread creation time, the priority is set to that of the process. The relative thread priorities are in a range of ± 2 “points” from the process’s base. The symbolic names of the resulting common thread priorities, starting with the five relative priorities, are:

- `THREAD_PRIORITY_LOWEST`
- `THREAD_PRIORITY_BELOW_NORMAL`
- `THREAD_PRIORITY_NORMAL`
- `THREAD_PRIORITY_ABOVE_NORMAL`
- `THREAD_PRIORITY_HIGHEST`
- `THREAD_PRIORITY_TIME_CRITICAL` is 15, or 31 if the process class is `REALTIME_PRIORITY_CLASS`.
- `THREAD_PRIORITY_IDLE` is 1, or 16 for `REALTIME_PRIORITY_CLASS` processes.
- `THREAD_MODE_BACKGROUND_BEGIN` and `THREAD_MODE_BACKGROUND_END` are similar to `PROCESS_MODE_BACKGROUND_BEGIN` and `PROCESS_MODE_BACKGROUND_END`. You need Windows Vista, or later, to use these modes.

Use these values to set and read a thread’s relative priority. Note the signed integer priority argument.

```

BOOL SetThreadPriority (HANDLE hThread,
    int nPriority)

int GetThreadPriority (HANDLE hThread)

```

There are actually two additional thread priority values. They are absolute rather than relative and are used only in special cases.

- `THREAD_PRIORITY_IDLE` is a value of 1 (or 16 for real-time processes).
- `THREAD_PRIORITY_TIME_CRITICAL` is 15 (or 31 for real-time processes).

Thread priorities change automatically with process priority. In addition, Windows may adjust thread priorities dynamically on the basis of thread behavior. You can enable and disable this feature with the `SetThreadPriorityBoost` function.

Thread and Process Priority Cautions

Use high thread priorities and process priority classes with caution or, better yet, not at all, unless there is a proven requirement. Definitely avoid real-time priorities for normal user processes; our examples never use real-time priorities, and real-time applications are out of scope. Among other dangers, user threads may preempt threads in the executive.

Furthermore, everything that we say in the following chapters about the correctness of threaded programs assumes, without comment, that thread scheduling is *fair*. Fairness ensures that all threads will, eventually, run. Without fairness, a low-priority thread could hold resources required by a high-priority thread. *Thread starvation* and *priority inversion* are terms used to describe the defects that occur when scheduling is not fair.

Thread States

Figure 7–4, which is taken from Custer’s *Inside Windows NT*, page 210 (also see Russinovich, Solomon, and Ionescu), shows how the executive manages threads and shows the possible thread states. This figure also shows the effect of program actions. Such state diagrams are common to all multitasking OSs and help clarify how a thread is scheduled for execution and how a thread moves from one state to another.

Here is a quick summary of the fundamentals; see the references for more information.

- A thread is in the *running* state when it is actually running on a processor. More than one thread can be in the running state on a multiprocessor computer.
- The executive places a running thread in the *wait* state when the thread performs a wait on a nonsignaled handle, such as a thread or process handle, or on a synchronization object handle (Chapter 8). I/O operations will also wait for completion of a disk or other data transfer, and numerous other functions can cause waiting. It is common to say that a thread is *blocked*, or *sleeping*, when in the wait state.

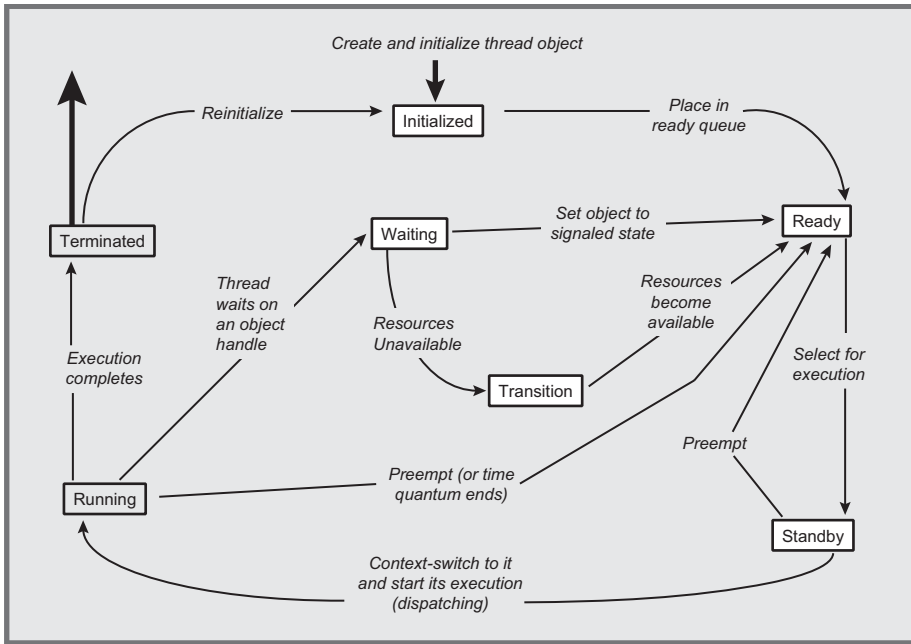


Figure 7-4 Thread States and Transitions

(From *Inside Windows NT*, by Helen Custer. Copyright © 1993, Microsoft Press. Reproduced by permission of Microsoft Press. All rights reserved.)

- A thread is *ready* if it could be running. The executive's scheduler could put it in the running state at any time. The scheduler will run the highest-priority ready thread when a processor becomes available, and it will run the one that has been in the ready state for the longest time if several threads have the same high priority. The thread moves through the *standby* state before entering the ready state.
- Normally, the scheduler will place a ready thread on any available processor. The programmer can specify a thread's *processor affinity* (see Chapter 9), which will limit the processors that can run that specific thread. In this way, the programmer can allocate processors to threads and prevent other threads from using these processors, helping to assure responsiveness for some threads. The appropriate functions are `SetProcessAffinityMask` and `GetProcessAffinityMask`. `SetThreadIdealProcessor` can specify a preferred processor that the scheduler will use whenever possible; this is less restrictive than assigning a thread to a single processor with the affinity mask.

- The executive will move a running thread to the ready state if the thread's time slice expires without the thread waiting. Executing `Sleep(0)` will also move a thread from the running state to the ready state.
- The executive will place a waiting thread in the ready state as soon as the appropriate handles are signaled, although the thread actually goes through an intermediate *transition* state. It is common to say that the thread *wakes up*.
- There is no way for a program to determine the state of another thread (of course, a thread, if it is running, must be in the running state, so it would be meaningless for a thread to find its own state). Even if there were, the state might change before the inquiring thread would be able to act on the information.
- A thread, regardless of its state, can be *suspended*, and a ready thread will not be run if it is suspended. If a running thread is suspended, either by itself or by a thread on a different processor, it is placed in the ready state.
- A thread is in the *terminated* state after it terminates and remains there as long as there are any open handles on the thread. This arrangement allows other threads to interrogate the thread's state and exit code.

Pitfalls and Common Mistakes

There are several factors to keep in mind as you develop threaded programs; lack of attention to a few basic principles can result in serious defects, and it is best to avoid the problems in the first place than try to find them during testing or debugging.

The essential factor is that the threads execute asynchronously. There is no sequencing unless you create it explicitly. This asynchronous behavior is what makes threads so useful, but without proper care, serious difficulties can occur.

Here are a few guidelines; there are more in later chapters. The example programs attempt to adhere to all these guidelines. There may be a few inadvertent violations, however, which illustrates the multithreaded programming challenges.

- Make no assumptions about the order in which the parent and child threads execute. It is possible for a child thread to run to completion before the parent returns from `CreateThread`, or, conversely, the child thread may not run at all for a considerable period of time. On a multiprocessor computer, the parent and one or more children may even run concurrently.
- Ensure that all initialization required by the child is complete before the `CreateThread` call, or else use thread suspension or some other technique. Failure by the parent to initialize data required by the child is a common cause of "race conditions" wherein the parent "races" the child to initialize data before the child needs it. `sortMT` illustrates this principle.

- Be certain that each distinct child has its own data structure passed through the thread function's parameter. Do not assume that one child thread will complete before another (this is another form of race condition).
- Any thread, at any time, can be preempted, and any thread, at any time, may resume execution.
- Do not use thread priority as a substitute for explicit synchronization.
- Do not use reasoning such as “that will hardly ever happen” as an argument that a program is correct. If it can happen, it will, possibly at a very embarrassing moment.
- Even more so than with single-threaded programs, testing is necessary, but not sufficient, to ensure program correctness. It is common for a multithreaded program to pass extensive tests despite code defects. There is no substitute for careful design, implementation, and code inspection.
- Threaded program behavior varies widely with processor speed, number of processors, OS version, and more. Testing on a variety of systems can isolate numerous defects, but the preceding precaution still applies.
- Be certain that threads have a sufficiently large stack, although the default 1MB is usually sufficient.
- Threads should be used only as appropriate. Thus, if there are activities that are naturally concurrent, each such activity can be represented by a thread. If, on the other hand, the activities are naturally sequential, threads only add complexity and performance overhead.
- If you use a large number of threads, be careful, as the numerous stacks will consume virtual memory space and thread context switching may become expensive. “Large” is a relative term and could mean hundreds or thousands. In other cases, it could mean more threads than the number of processors.
- Fortunately, correct programs are frequently the simplest and have the most elegant designs. Avoid complexity wherever possible.

Timed Waits

The final function, `Sleep`, allows a thread to give up the processor and move from the running to the wait state for a specified period of time. A thread can, for example, perform a task periodically by sleeping after carrying out the task. Once the time period is over, the scheduler moves the thread back to the ready state. A program in Chapter 11 (Program 11–4) uses this technique.