```
VOID Sleep (DWORD dwMilliseconds)
```

The time period is in milliseconds and can even be `INFINITE`, in which case the thread will never resume. A `0` value will cause the thread to relinquish the remainder of the time slice; the kernel moves the thread from the running state to the ready state (Figure 7–4).

The function `SwitchToThread` provides another way for a thread to yield its processor to another ready thread if there is one that is ready to run.

The UNIX `sleep` function is similar to `Sleep`, but time periods are measured in seconds. To obtain millisecond resolution, use the `select` or `poll` functions with no file descriptors.

Fibers

Note: Fibers are of specialized interest. See the comment after the first bulleted item below to determine if you want to skip this section.

A *fiber,* as the name implies, is a piece of a thread. More precisely, a fiber is a unit of execution that can be scheduled by the application rather than by the kernel. An application can create numerous fibers, and the fibers themselves determine which fiber will execute next. The fibers have independent stacks but otherwise run entirely in the context of the thread on which they are scheduled, having access, for example, to the thread's TLS and any mutexes[4] owned by the thread. Furthermore, fiber management occurs entirely in user space outside the kernel. Fibers can be thought of as lightweight threads, although there are numerous differences.

A fiber can execute on any thread, but never on two at one time. Therefore, a fiber should not access thread-specific data, such as TLS, as the data will have no meaning if the fiber is later rescheduled to run on another thread.

Fibers can be used for several purposes.

- Most importantly, many applications, especially some written for UNIX using proprietary thread implementations, now generally obsolete, are written to schedule their own threads. Fibers make it easier to port such applications to Windows but otherwise do not provide advantages over properly used threads. *Most readers will not have such requirements and may want to skip this section.*

[4] A mutex, as explained in Chapter 8, is a synchronization object that threads can own.

- A fiber does not need to block waiting for a file lock, mutex, named pipe input, or other resource. Rather, one fiber can poll the resource and, if the resource is not available, switch control to another specific fiber.
- Fibers operate as part of a *converted* thread (see the first numbered item below) and have access to thread and process resources. A fiber is not, however, bound to a specific thread and can run on any thread (but not on more than one at a time).
- Unlike threads, fibers are not preemptively scheduled. The Windows executive, in fact, is not aware of fibers; fibers are managed within the fiber DLL entirely within user space.
- Fibers allow you to implement *co-routines*, whereby an application switches among several interrelated tasks. Threads do not allow this. The programmer has no direct control over which thread will be executed next.
- Major software vendors have used fibers and claim performance advantages. For example, Oracle Database 10g has an optional "fiber mode" (see http://download.oracle.com/owsf_2003/40171_colello.ppt; this presentation also describes the threading model).

Seven functions make up the fiber API. They are used in the following sequence and as shown in Figure 7–5.

1. A thread must first enable fiber operation by calling `ConvertThreadToFiber` or `ConvertThreadToFiberEx`. The thread then consists of a single fiber. This call provides a pointer to fiber data, which can be used in much the same way that the thread argument was used to create unique data for a thread.
2. The application can create additional fibers using `CreateFiber`. Each fiber has a start address, a stack size, and a parameter. Each new fiber is identified by an address rather than by a handle.
3. An individual fiber can obtain its data, as received from `CreateFiber`, by calling `GetFiberData`.
4. Similarly, a fiber can obtain its identity with `GetCurrentFiber`.
5. A running fiber yields control to another fiber by calling `SwitchToFiber`, indicating the address of the other fiber. Fibers must explicitly indicate the next fiber that is to run within the thread.
6. The `DeleteFiber` function deletes an existing fiber and all its associated data.

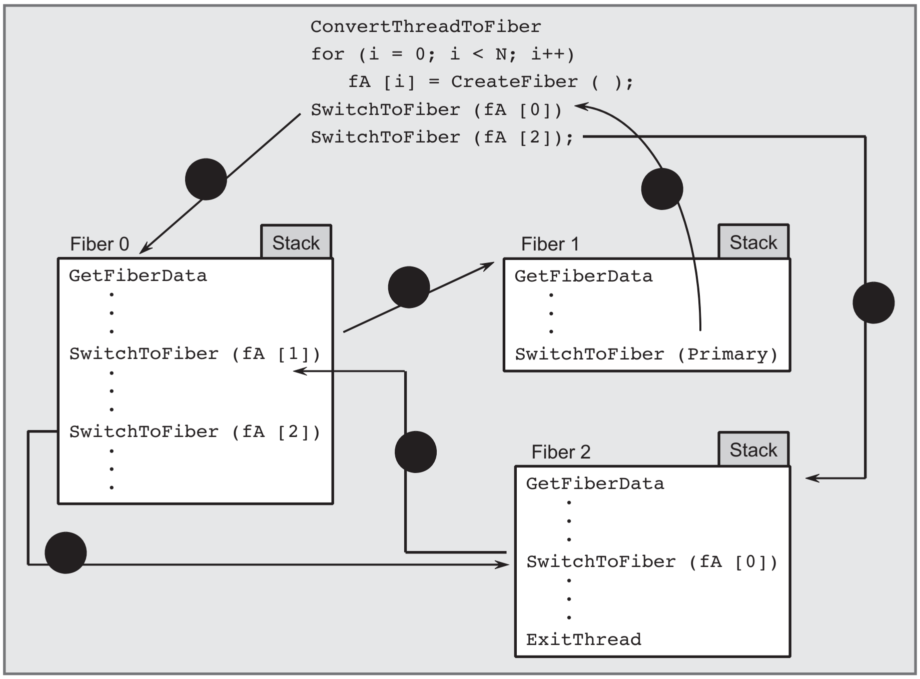

Figure 7–5 Control Flow among Fibers in a Thread

7. New functions, such as `ConvertFiberToThread` (which releases resources created by `ConvertThreadToFiber`), have been added to XP (NT 5.1), along with fiber local storage.

Figure 7–5 shows fibers in a thread. This example shows two ways in which fibers schedule each other.

- ***Master-slave* scheduling**. One fiber decides which fiber to run, and that fiber always yields control to the master fiber. Fiber 1 in Figure 7–5 behaves in this way. The *Examples* file contains `grepMF`, a `grepMT` variation, that uses master-slave scheduling.
- ***Peer-to-peer* scheduling**. A fiber determines the next fiber to run. The determination can be based on policies such as round-robin scheduling, priority scheduling based on a priority scheme, and so on. Co-routines would be implemented with peer-to-peer scheduling. In Figure 7–5, Fibers 0 and 2 switch control in this way.

Summary

Windows supports threads that are independently scheduled but share the same process address space and resources. Threads give the programmer an opportunity to simplify program design and to exploit parallelism in the application to improve performance. Threads can even yield performance benefits on single-processor systems. Fibers are units of execution that the program, rather than the Windows executive, schedules for execution.

Looking Ahead

Chapter 8 describes and compares the Windows synchronization objects, and Chapters 9 and 10 continue with more advanced synchronization topics, performance comparisons, and extended examples. Chapter 11 implements the threaded server shown in Figure 7–1.

Additional Reading

Windows

Multithreading Applications in Win32, by Jim Beveridge and Robert Wiener, is an entire book devoted to Win32 threads.

UNIX and Pthreads

Both *Advanced Programming in the UNIX Environment,* by W. Richard Stevens and Stephen A. Rago, and *Programming with POSIX Threads,* by David Butenhof, are recommended. The second book provides numerous guidelines for threaded program design and implementation. The information applies to Windows as well as to Pthreads, and many of the examples can be easily ported to Windows. There is also good coverage of the boss/worker, client/server, and pipeline threading models, and Butenhof's presentation is the basis for the model descriptions in this chapter.

Exercises

7–1. Implement a set of functions that will suspend and resume threads but also allow you to obtain a thread's suspend count.

7–2. Compare the performance of the parallel word count programs, one using threads (`wcMT`) and the other using processes (similar to Program 6–1, `grepMP`). Compare the results with those in Appendix C.

7–3. Perform additional performance studies with `grepMT` where the files are on different disk drives or are networked files. Also determine the performance gain on as many multiprocessor systems as are available.

7–4. Modify `grepMT`, Program 7–1, so that it puts out the results in the same order as that of the files on the command line. Does this affect the performance measurements in any way?

7–5. Further enhance `grepMT`, Program 7–1, so that it prints the time required by each worker thread. `GetThreadTimes` will be useful, and this function is similar to `GetProcessTimes` (Chapter 6).

7–6. The *Examples* file includes a multithreaded word count program, `wcMT.c`, that has a structure similar to that of `grepMT.c`. A defective version, `wcMTx.c`, is also included. Without referring to the correct solution, analyze and fix the defects in `wcMTx.c`, including any syntax errors. Also, create test cases that illustrate these defects and carry out performance experiments similar to those suggested for `grepMT`. If you use Cygwin (open source UNIX/Linux commands and shells), compare the performance of Cygwin's `wc` with that of `wcMT`, especially on multiprocessor systems.

7–7. The *Examples* file includes `grepMTx.c`, which is defective because it violates basic rules for thread safety. Describe the failure symptoms, identify the errors, and fix them.

7–8. `sortMT` requires that the number of records in the array to be sorted be divisible by the number of threads and that the number of threads be a power of 2. Remove these restrictions.

7–9. Enhance `sortMT` so that if the number of threads specified on the command line is zero, the program will determine the number of processors on the host computer using `GetSystemInfo`. Set the number of threads to different multiples (1, 2, 4, and so on) of the number of processors and determine the effect on performance.

7–10. Modify `sortMT` so that the worker threads are not suspended when they are created. What failure symptoms, if any, does the program demonstrate as a result of the race condition defect?

7–11. `sortMT` reads the entire file in the primary thread before creating the sorting threads. Modify the program so that each thread reads the portion of the file that it requires. Next, modify the program to use mapped files.

7–12. Is there any performance benefit if you give some of the threads in `sortMT` higher priority than others? For example, it might be beneficial to give the threads that only sort and do not merge, such as Thread 3 in Figure 7–2, a higher priority. Explain the results.

7–13. `sortMT` creates all the threads in a suspended state so as to avoid a race condition. Modify the program so that it creates the threads in reverse order and in a running state. Are there any remaining race conditions? Compare performance with the original version.

7–14. Quicksort, the algorithm generally used by the C library `qsort` function, is usually fast, but it can be slow in certain cases. Most texts on algorithms show a version that is fastest when the array is reverse sorted and slowest when it is already sorted. The Microsoft C library implementation is different. Determine from the library code which sequences will produce the best and worst behavior, and study `sortMT`'s performance in these extreme cases. What is the effect of increasing or decreasing the number of threads? *Note:* The C library source code can be installed in the `CRT` directory under your Visual Studio installation. Look for `qsort.c`.

7–15. The *Examples* file contains a defective `sortMTx.c` program. Demonstrate the defects with test cases and then explain and fix the defects without reference to the correct solutions. *Caution:* The defective version may have syntax errors as well as errors in the thread logic.

7–16. One of the technical reviewers suggested an interesting `sortMT` enhancement that may provide improved performance. The idea is to modify the `MergeArrays` function so that it does not need to allocate the destination record storage. Instead, preallocate a second array as large as the array being sorted. Each worker thread can then use the appropriate portion of the preallocated array. Finally, eliminate the `memcpy` at the end. *Hint:* Alternate the merge direction on even and odd passes. Compare the resulting performance to Runs 7–2a and 7–2b.

CHAPTER

8 Thread Synchronization

Threads can simplify program design and implementation and also improve performance, but thread usage requires care to ensure that shared resources are protected against simultaneous modification and that threads run only when appropriate. This chapter shows how to use Windows synchronization objects—`CRITICAL_SECTION`s, mutexes, semaphores, and events[1]—to solve these problems and describes some of the problems, such as deadlocks and race conditions, that can occur with improper synchronization object use. Some synchronization objects can be used to synchronize threads in the same process or in separate processes.

The examples illustrate the synchronization objects and discuss the performance impacts, both positive and negative, of different synchronization methods. The following chapters then show how to use synchronization to solve additional programming problems, improve performance, avoid pitfalls, and use more advanced NT6 features, such as "slim reader/writer" (SRW) locks and Windows condition variables.

Thread synchronization is a fundamental and interesting topic, and it is essential in nearly all threaded applications. *Nonetheless, readers who are primarily interested in interprocess communication, network programming, and building threaded servers might want to skip to Chapter 11.*

The Need for Thread Synchronization

Chapter 7 showed how to create and manage worker threads, where each worker thread accesses its own resources and runs to completion without interacting with other threads. Each thread in the Chapter 7 examples processes a separate file or a separate storage area, yet simple synchronization during thread creation and termination is still necessary. For example, the `grepMT` worker threads all run

[1] The last three are Windows *kernel* objects referenced with `HANDLE`s. The first is not a kernel object.

independently of one another, but the boss thread must wait for the workers to complete before reporting the results the worker threads generated. Notice that the boss shares memory with the workers, but the program design assures that the boss will not access the memory until the worker terminates. This design enables the parallelism described in Chapter 7.

`sortMT` is slightly more complicated because the workers need to synchronize by waiting for adjacent workers to complete, and the worker threads are not allowed to start until the boss thread has created all the workers. As with `grepMT`, synchronization consists of waiting for one or more threads to terminate.

In many cases, however, it is necessary for two or more threads to coordinate execution throughout each thread's lifetime. For instance, several threads may share data, and this raises the issue of mutual exclusion. In other cases, a thread cannot proceed until another thread reaches a designated point. How can the programmer assume that two or more threads do not, for example, simultaneously modify the same global storage, such as the performance statistics? Furthermore, how can the programmer ensure that a thread does not attempt to remove an element from a queue before there are any elements in the queue or that two threads do not attempt to remove the same element?

Several examples illustrate situations that can prevent code from being thread-safe. (Code is thread-safe if several threads can execute the code simultaneously without any undesirable results.) Thread safety is discussed later in this and the following chapters.

Figure 8–1 shows what can happen when two unsynchronized threads share a resource such as a memory location. Both threads increment variable `N`, but, because of the particular sequence in which the threads *might* execute, the final value of `N` is `5`, whereas the correct value is `6`. Notice that the particular result shown here is not predictable; a different thread execution sequence could yield the correct results. Execution on a multiprocessor computer can aggravate this problem.

Critical Code Regions

Incrementing `N` with a single statement such as `N++` is no better because the compiler will generate a sequence of one or more machine-level instructions that are not necessarily executed *atomically* as a single unit.

The core problem is that there is a *critical code region*[2] (the code that increments `N` in this example) such that, once a thread starts to execute the critical region, no other thread can be allowed to enter until the first thread exits

[2] The term "critical code section" is common but can cause confusion with Windows `CRITICAL_SECTION` objects, which, while related to critical code regions (or sections), are not the same thing.

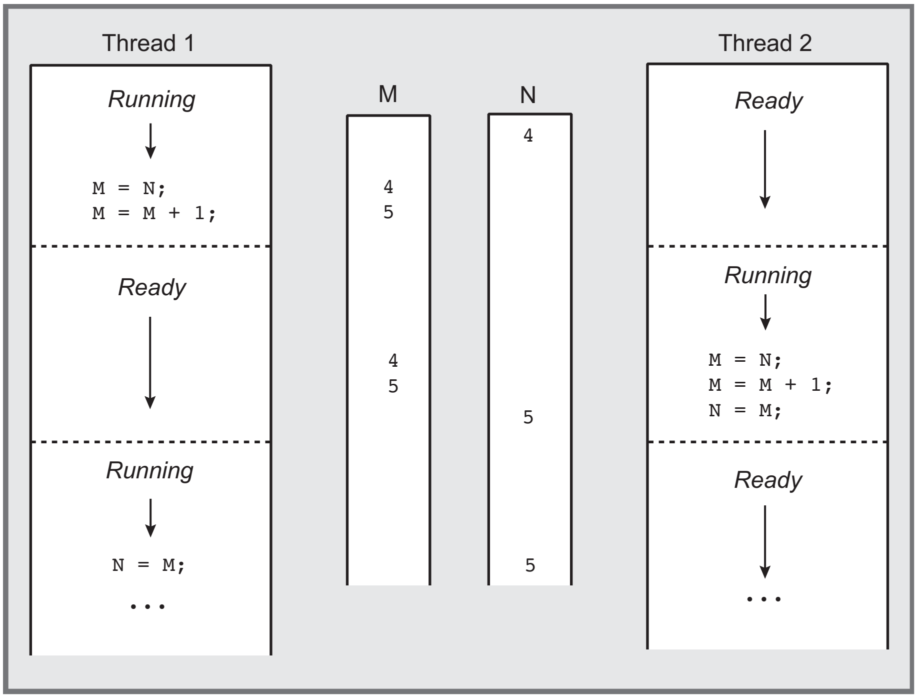

Figure 8–1 Unsynchronized Threads Sharing Memory

from the code region. This critical code region problem can be considered a type of race condition because the first thread "races" to complete the critical region before any other thread starts to execute the same critical code region. Thus, we need to synchronize thread execution in order to ensure that only one thread at a time executes the critical region.

There can be more than one critical code region for a variable, such as `N` in Figure 8–1. Typically, there might be a critical code region that decrements `N`. Generalizing, we need to synchronize thread execution in order to ensure that only one thread at a time executes *any* of the critical regions for a data item. We need to avoid problems such as having one thread increment `N` while another is decrementing it.

Defective Solutions to the Critical Code Region Problem

Similarly unpredictable results will occur with a code sequence that attempts to protect the increment with a global polled flag (in this case, the variable `Flag`).

```
BOOL Flag = FALSE;
DWORD N;
. . .
DWORD WINAPI ThreadFunc(TH_ARGS pArgs)
{
   . . .
   while (Flag) Sleep (1000);
   Flag = TRUE;
   N++;
   Flag = FALSE;
   . . .
}
```

Even in this case, the thread could be preempted between the time `Flag` is tested and the time `Flag` is set to `TRUE`; the first two statements form a critical code region that is not properly protected from concurrent access by two or more threads.

Another attempted solution to the critical code region synchronization problem might be to give each thread its own copy of the variable `N`, as follows:

```
DWORD WINAPI ThreadFunc (TH_ARGS pArgs)
{  DWORD N;
   ... N++; ...
}
```

This approach is no better, however, because each thread has its own copy of the variable on its stack, where it may have been required to have `N` represent, for example, the total number of threads in operation. Such a solution is necessary, however, in the common case in which each thread needs its own distinct copy of the variable and the increment is not a critical code region.

Notice that such problems are not limited to threads within a single process. They can also occur if two processes share mapped memory or modify the same file.

`volatile` Storage

Yet another latent defect exists even after we solve the synchronization problem. An optimizing compiler might leave the value of `N` in a register rather than storing it back in `N`. An attempt to solve this problem by resetting compiler optimization switches would impact performance throughout the code. The correct solution is to use the ANSI C `volatile` storage qualifier, which ensures that the variable will be stored in memory after modification and will always be fetched from memory before use. The `volatile` qualifier informs the compiler that the variable can change value at any time. Be aware, however, that the `volatile` qualifier can negatively affect performance, so use it only as required.

As a simple guideline, use `volatile` for any variable that is accessed by concurrent threads and is:

- Modified by at least one thread, and
- Accessed, even if read-only, by two or more threads, and correct program operation depends on the new value being visible to all threads immediately

This guideline is overly cautious; Program 8–1 shows a situation where the variable meeting these guidelines does not necessarily need to be `volatile`. If a modifying thread returns from or calls another function after modifying the variable, the variable will not be held in a register.

There is another situation where you need to use `volatile`; the parameters to the "interlocked functions," described soon, require `volatile` variables.

Memory Architecture and Memory Barriers

Even the `volatile` modifier does not assure that changes are visible to other processors in a specific order, because a processor might hold the value in cache before committing it to main memory and alter the order in which different processes see the changed values. To assure that changes are visible to other processors in the desired order, use *memory barriers* (or "fences"); the interlocked functions (next section) provide a memory barrier, as do all the synchronization functions in this chapter.

To help clarify this complex issue, Figure 8–2 shows the memory subsystem architecture of a typical multiprocessor computer. In this case, the computer has four processors on two dual-core chips and is similar to the computer used with many of the "run" screenshots in this chapter and Chapter 7.

The components are listed here, along with *representative* values[3] for total size, line size (that is, the number of bytes in a single chunk), and latency (access) times in processor cycles:

- The four processor cores, which include the registers that hold computed values as well as values loaded from memory.
- Level 1 (L1) cache. The instruction and data caches are usually separate, and each processor core has a distinct cache. When you modify a `volatile` vari-

[3] See the chip manufacturer's specifications for actual values and architectural details. The information here is similar to that of the Intel Core 2 Quad processor.

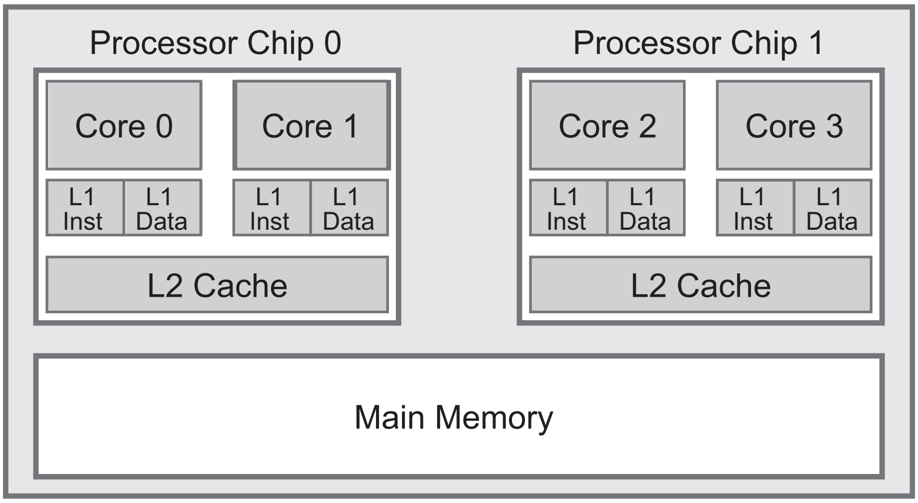

Figure 8–2 Memory System Architecture

able, the new value will be stored in the core's L1 data cache but won't necessarily be stored in the L2 cache or main memory. *Size:* 32KB, *Line Size:* 64 bytes, *Latency:* 3 cycles.

- Level 2 (L2) cache. Each processor chip has its own L2 cache, shared by the two cores. *Size:* 6MB, *Line Size:* 64 bytes, *Latency:* 14 cycles.
- Main Memory, which is shared by all processor cores and is not part of the processor chips. *Size:* Multiple GB, *Line Size:* N/A, *Latency:* 100+ cycles.

Figure 8–2 represents the most common "symmetric multiprocessing" (SMP) shared memory architecture, although the processors are not entirely symmetric because of the L2 cache. Nonuniform memory access (NUMA) is more complex because the main memory is partitioned among the processors; NUMA is not coverd here.

Figure 8–2 shows that `volatile` only assures that the new data value will be in the L1 cache; there is no assurance that the new value will be visible to threads running on other cores. A memory barrier, however, assures that the value is moved to main memory. Furthermore, the barrier assures cache coherence. Thus, if Core 0 updates variable `N` at a memory barrier, and Core 3's L1 cache has a value representing `N`, the `N` value in Core 3's L1 (and L2) cache is either updated or removed so that the new value is visible to Core 3 and all other cores concurrently.

There is a performance cost, however, as moving data between main memory, processor cores, and caches can require hundreds of cycles, whereas a pipelined processor can access register values in less than a cycle.