Figure 8–2 also shows that it's important to assure that shared variables are aligned on their natural boundaries. If, for example, a `LARGE_INTEGER` were aligned on a 4-byte (but not 8-byte) boundary, it might also cross a cache line boundary. It would then be possible that only part of the new value would become visible to other processors, resulting in a "word tear" bug. By default, most compilers align data items on their natural boundaries.

Interlocked Functions: Introduction

If all we need is to increment, decrement, or exchange variables, as in this simple initial example, then the *interlocked* functions will suffice, and the variables need to be `volatile`. The interlocked functions are simpler and faster than any of the alternatives, although they do generate a memory barrier with the performance impact described previously.

The first two members of the interlocked function family are `InterlockedIncrement` and `InterlockedDecrement`; other interlocked functions are described in a later section. These two instructions apply to 32-bit signed integers (the "Addend," which must be aligned on a 4-byte boundary to assure correct operation) and return the resulting `Addend` value.

These functions have limited utility, but they should be used wherever possible to simplify code and improve performance.

```
LONG InterlockedIncrement (
   LONG volatile *Addend)
```

```
LONG InterlockedDecrement (
   LONG volatile *Addend)
```

Use `InterlockedIncrement64` and `InterlockedDecrement64` to increment and decrement 64-bit values, but be sure that the `Addend` is aligned on a 64-bit (8-byte) boundary.

If your code will run on processors that support "acquire" and "release" semantics, such as the Itanium (but not Intel x86 and x64), you could use `InterlockedIncrementAcquire` and `InterlockedIncrementRelease` to gain performance. See MSDN for more information.

The task of incrementing N in Figure 8–1 can be implemented with a single line:

```
InterlockedIncrement (&N);
```

N is a signed volatile LONG integer, and the function returns its new value, although another thread could modify N's value before the thread that called InterlockedIncrement can use the returned value.

Be careful, however, not to call this function twice in succession if, for example, you need to increment the variable by 2 and correct program operation depends on the variable being even. The thread might be preempted between the two calls. Instead, use the InterlockedExchangeAdd function described near the end of the chapter.

Local and Global Storage

Another requirement for correct thread code is that global storage not be used for local purposes. For example, the earlier ThreadFunc example would be necessary and appropriate if each thread required its own separate copy of N. N might hold temporary results or retain the argument. If, however, N represents thread-specific data and were placed in global storage, all threads would share a single copy of N, resulting in incorrect behavior no matter how well your program synchronized access.

Here is an example of such incorrect usage, which often occurs when converting a legacy single-threaded program to multithreaded operation and using a function (ThreadFunc, in this case) as a thread function. N should be a local variable, allocated on the thread function's stack as its value is used within the function.

```
DWORD N;
. . .
DWORD WINAPI ThreadFunc (TH_ARGS pArgs)
{
   ...
   N = 2 * pArgs->Count; ...
   /* Use N; value is specific to this call to ThreadFunc */
}
```

Comment: Finding and removing global variables, such as N in this fragment, is a major challenge when converting legacy, single-threaded programs to use threads. In the code fragment above, the function was called sequentially, and, in the multithreaded version, several threads can be executing the function concurrently. The problem is also challenging when we have a situation such as the following legacy code fragment, where results are accumulated in the global variable:

```
DWORD N = 0;
. . .
for (int i = 0; i < MAX; ++i) {
   ... Allocate and initialize pArgs data ...
   N += ThreadFunc(pArgs);
}
. . .
DWORD WINAPI ThreadFunc (ARGS pArgs)
{
   DWORD result;
   ...
   result = ...;
   return result;
}
```

The challenge occurs when converting `ThreadFunc` to a thread function executed by two or more threads running in parallel. This and the following chapters deal with many similar situations.

Summary: Thread-Safe Code

Before proceeding to the synchronization objects, here are five initial guidelines to help ensure that the code will run correctly in a threaded environment.

1. Variables that are local to the thread should not be global and should be on the thread's stack or in a data structure or TLS that only the individual thread can access directly.
2. If a function is called by several threads and a thread-specific state value, such as a counter, is to persist from one function call to the next, do not store it in a global variable or structure. Instead, store the state value in TLS or in a data structure dedicated to that thread, such as the data structure passed to the thread when it is created. Programs 12–5 and 12–6 show the required techniques when building thread-safe DLLs.
3. Avoid race conditions such as the uninitialized variables that would occur in Program 7–2 (`sortMT`) if the threads were not created in a suspended state. If some condition is assumed to hold at a specific point in the program, wait on a synchronization object to ensure that the condition does hold.

4. Threads should not, in general, change the process environment because that would affect all threads. Thus, a thread should not set the standard input or output handles or change environment variables. An exception would be the primary thread, which might make such changes before creating any other threads. In this case, all threads would share the same environment, since the primary thread can assure that there are no other threads in the process at the time the environment is changed.
5. Variables shared by all threads should be static or in global storage and protected with the synchronization or interlocked mechanisms that create a memory barrier.

The next section discusses the synchronization objects. With that discussion, there will be enough to develop a simple producer/consumer example.

Thread Synchronization Objects

Two mechanisms discussed so far allow processes and threads to synchronize with one another.

1. A thread can wait for another process to terminate by waiting on the process handle with `WaitForSingleObject` or `WaitForMultipleObjects`. A thread can wait for another thread to terminate, regardless of how the thread terminates, in the same way.
2. File locks are specifically for synchronizing file access.

Windows NT5 and NT6 provide four other objects designed for thread and process synchronization. Three of these objects—*mutexes*, *semaphores*, and *events*—are kernel objects that have handles. Events are also used for other purposes, such as asynchronous I/O (Chapter 14).

The fourth object, the `CRITICAL_SECTION`, is discussed first. Because of their simplicity and performance advantages, `CRITICAL_SECTION`s are the preferred mechanism when they are adequate for a program's requirements.

Caution: There are risks inherent to the use of synchronization objects if they are not used properly. These risks, such as deadlocks, are described in this and subsequent chapters, along with techniques for developing reliable code. First, however, we'll show some synchronization examples in realistic situations.

New in Windows Vista and Windows Server 2008: Windows kernel 6 (NT 6) introduced SRW locks (see Chapter 9) and condition variables (see Chapter 10), which are welcome additions. However, at the time of writing, most applications will need to support Windows XP. This situation may change in the future.

Two other synchronization objects, waitable timers and I/O completion ports, are deferred until we've described the prerequisite asynchronous I/O techniques in Chapter 14.

CRITICAL_SECTION Objects

A critical code region, as described earlier, is a code region that only one thread can execute at a time; more than one thread executing the critical code region concurrently can result in unpredictable and incorrect results.

Windows provides the `CRITICAL_SECTION` object as a simple "lock" mechanism for implementing and enforcing the critical code region concept.

`CRITICAL_SECTION` (CS) objects are initialized and deleted but do not have handles and are not shared with other processes. Declare a CS variable as a `CRITICAL_SECTION`. Threads enter and leave a CS, and only one thread at a time can be in a specific CS. A thread can, however, enter and leave a specific CS at multiple points in the program.

To initialize and delete a `CRITICAL_SECTION` variable and its resources, use `InitializeCriticalSection` and `DeleteCriticalSection`, respectively. You cannot perform any operations on a CS before initializing it or after deleting it, although you can reinitialize a CS.

```
VOID InitializeCriticalSection (
   LPCRITICAL_SECTION lpCriticalSection)
```

```
VOID DeleteCriticalSection (
   LPCRITICAL_SECTION lpCriticalSection)
```

`EnterCriticalSection` blocks a thread if another thread is in the section, and multiple threads can wait simultaneously on the same CS. One waiting thread unblocks when another thread executes `LeaveCriticalSection`; you cannot predict which waiting thread will unblock.

We say that a thread *owns* the CS once it returns from `EnterCritical-Section`, and `LeaveCriticalSection` relinquishes ownership. *Always be certain to leave a CS; failure to do so will cause other threads to wait forever, even if the owning thread terminates.* The examples use `__finally` blocks to leave CSs.

We will often say that a CS is *locked* or *unlocked,* and entering a CS is the same as locking the CS.

```
VOID EnterCriticalSection (
   LPCRITICAL_SECTION lpCriticalSection)

VOID LeaveCriticalSection (
   LPCRITICAL_SECTION lpCriticalSection)
```

If a thread already owns the CS, it can enter again without blocking; that is, `CRITICAL_SECTION`s are *recursive.* Windows maintains a count so that the thread must leave as many times as it enters in order to unlock the CS for other threads. This capability can be useful in implementing recursive functions and making shared library functions thread-safe.

Leaving a CS that a thread does not own can produce unpredictable results, including thread blockage.

There is no time-out from `EnterCriticalSection`; a thread will block forever if the owning thread never leaves the CS. You can, however, test or poll to see whether another thread owns a CS using `TryEnterCriticalSection`.

```
BOOL TryEnterCriticalSection (
   LPCRITICAL_SECTION lpCriticalSection)
```

A `TRUE` return value from `TryEnterCriticalSection` indicates that the calling thread now owns the CS. A `FALSE` return indicates that some other thread already owns the CS, and it is not safe to execute the critical code region.

`CRITICAL_SECTION`s have the advantage of not being kernel objects and are maintained in user space. This almost always provides performance improvements compared to using a Windows mutex kernel object with similar functionality, especially in NT5 and later (and this book assumes you are using NT5 or NT6). We will discuss the performance benefit after introducing kernel synchronization objects.

Adjusting the Spin Count

Normally, if a thread finds that a CS is already owned when executing `EnterCriticalSection`, it enters the kernel and blocks until the `CRITICAL_SECTION` is released, which is time consuming. On multiprocessor systems, however, you can require that the thread try again (that is, spin) before blocking, as the owning thread may be running on a separate processor and could release the CS at any time. This can be useful for performance when there is high contention among threads for a single `CRITICAL_SECTION` that is never held for more than a few instructions. Performance implications are discussed later in this chapter and the next.

The two functions to adjust spin count are `SetCriticalSectionSpinCount`, which allows you to adjust the count dynamically, and `InitializeCriticalSectionAndSpinCount`, which is a substitute for `InitializeCriticalSection`. Spin count tuning is a topic in Chapter 9.

A `CRITICAL_SECTION` for Protecting Shared Variables

Using `CRITICAL_SECTION`s is simple, and one common use is to allow threads to access global shared variables. For example, consider a threaded server (as in Figure 7–1) in which there might be a need to maintain usage statistics such as:

- The total number of requests received
- The total number of responses sent
- The number of requests currently being processed by server threads

Because the count variables are global to the process, two threads must not modify the counts simultaneously. `CRITICAL_SECTION` objects provide one means of ensuring this, as shown by the code sequence below and in Figure 8–3. Program 8–1, much simpler than the server system, illustrates this `CRITICAL_SECTION` usage.

CSs can be used to solve problems such as the one shown in Figure 8–1, in which two threads increment the same variable. The following code segment will do more than increment the variable because simple incrementing is possible with the interlocked functions. This example also uses an intermediate variable; this unnecessary inefficiency more clearly illustrates the solution to the problem in Figure 8–1.

```
CRITICAL_SECTION cs1;
volatile DWORD N = 0;
```

```
/* N is a global variable, shared by all threads. */
InitializeCriticalSection (&cs1);
    . . .
/* Create one or more threads using ThreadFunc */
    . . .
DWORD WINAPI ThreadFunc (TH_ARGS pArgs);
{
    DWORD M;
    __try {
        EnterCriticalSection (&cs1);
    if (N < N_MAX) { M = N; M += 1; N = M; }
    } __finally {
        LeaveCriticalSection (&cs1)
    }
}
    ...
DeleteCriticalSection (&cs1);
```

Figure 8–3 shows one possible execution sequence for the Figure 8–1 example and illustrates how CSs solve the critical code region synchronization problem.

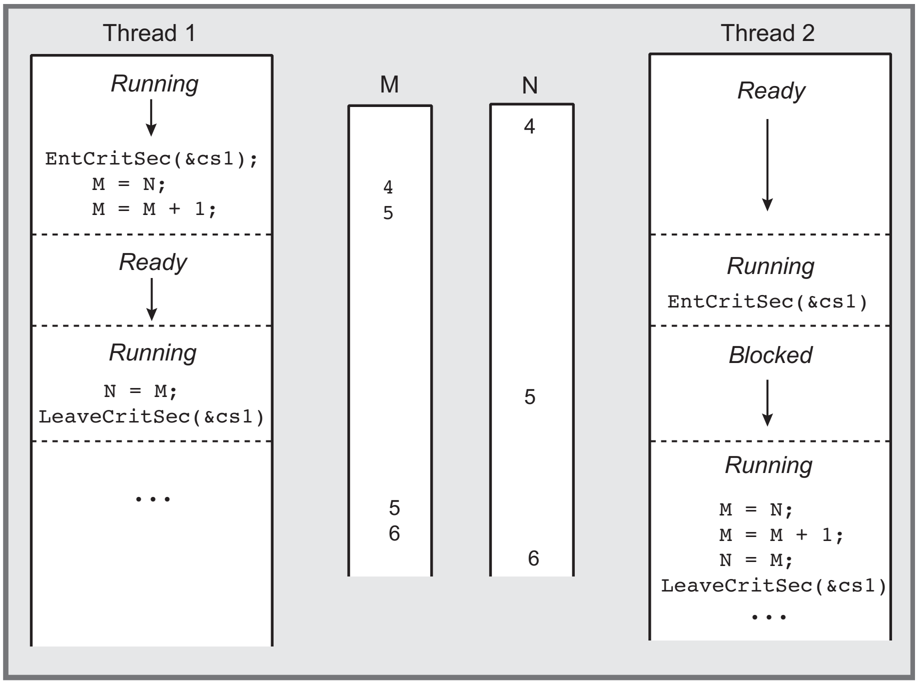

Figure 8–3 Synchronized Threads Sharing Memory

Protect a Variable with a Single Synchronization Object

Each variable, or collection of variables, that is accessed in a critical code section should be guarded by the *same* CS (or other synchronization object) everywhere. Otherwise, two threads could still modify the variable concurrently. For example, the following thread function, which uses two CS and interlocked functions, is defective (`N` is, as before, a global `volatile DWORD`, and `cs1` and `cs2` are global CSs).

```
DWORD WINAPI ThreadFunc (ARGS pArgs)
{
   . . .
   EnterCriticalSection(&cs1);
   N += 5;
   LeaveCriticalSection(&cs1);
   . . .
   InterlockedDecrement(&N);
   . . .
   EnterCriticalSection(&cs2);
   N -= 5;
   LeaveCriticalSection(&cs2);
   . . .
}
```

Example: A Simple Producer/Consumer System

Program 8–1 shows how CS lock objects can be useful. The program also shows how to build protected data structures for storing object state and introduces the concept of an *invariant*, which is a property of an object's state that is guaranteed (by the proper program implementation) to be true outside a critical code region. Here is a description of the problem.

- There are two threads in addition to the primary thread, a *producer* and a *consumer*, that act entirely asynchronously.
- The producer periodically creates messages containing a table of numbers, such as current stock prices, periodically updating the table.
- The consumer, on request from the user, displays the current data. The requirement is that the displayed data must be the *most recent complete set of data, but no data should be displayed twice.*

- Do not display data while the producer is updating it, and do not display old data. Note that, by design, many produced messages are never used and are "lost." This example is a special case of the pipeline model in which data moves from one thread to the next.
- As an integrity check, the producer also computes a simple checksum[4] of the data in the table, and the consumer validates the checksum to ensure that the data has not been corrupted in transmission from one thread to the next. If the consumer accesses the table while it is still being updated, the table will be invalid; the CS ensures that this does not happen. The message block invariant is that the checksum is correct for the current message contents.
- The two threads also maintain statistics on the total number of messages produced, consumed, and lost.

The final output (see Run 8–1 after Program 8–1) shows the actual number computed at stop time, since the number computed after a consume command is probably stale. Additional comments follow the program listing and the run screenshot.

Program 8–1 `simplePC:` A Simple Producer and Consumer

```
/* Chapter 8. simplePC.c */
/* Maintain two threads, a producer and a consumer. */
/* The producer periodically creates checksummed data buffers, */
/* or "message blocks," that the consumer displays when prompted. */

#include "Everything.h"
#include <time.h>
#define DATA_SIZE 256

typedef struct MSG_BLOCK_TAG { /* Message block */
   CRITICAL_SECTION mGuard;/* Guard the message block structure*/
   DWORD fReady, fStop;
      /* ready state flag, stop flag*/
   volatile DWORD nCons, mSequence; /* Msg block sequence number*/
   DWORD nLost;
   time_t mTimestamp;
   DWORD mChecksum; /* Message contents mChecksum*/
   DWORD mData[DATA_SIZE]; /* Message Contents*/
} MSG_BLOCK;

/* Single message block, ready to fill with a new message. */
MSG_BLOCK mBlock = { 0, 0, 0, 0, 0 };
```

[4] This checksum, an "exclusive or" of the message bits, is for illustration only. Much more sophisticated message digest techniques are available for use in production applications.

```
DWORD WINAPI Produce (void *);
DWORD WINAPI Consume (void *);
void MessageFill (MSG_BLOCK *);
void MessageDisplay (MSG_BLOCK *);

DWORD _tmain (DWORD argc, LPTSTR argv [])
{
   DWORD status;
   HANDLE hProduce, hConsume;

   /* Initialize the message block CRITICAL SECTION */
   InitializeCriticalSection (&mBlock.mGuard);

   /* Create the two threads */
   hProduce = (HANDLE)_beginthreadex (NULL, 0, Produce,
         NULL, 0, NULL);
   hConsume = (HANDLE)_beginthreadex (NULL, 0, Consume,
         NULL, 0, NULL);

   /* Wait for the producer and consumer to complete */

   status = WaitForSingleObject (hConsume, INFINITE);
   status = WaitForSingleObject (hProduce, INFINITE);

   DeleteCriticalSection (&mBlock.mGuard);

   _tprintf (_T("Producer and consumer threads have terminated\n"));
   _tprintf (_T("Messages produced: %d, Consumed: %d, Lost: %d.\n"),
      mBlock.mSequence, mBlock.nCons,
      mBlock.mSequence - mBlock.nCons);
   return 0;
}

DWORD WINAPI Produce (void *arg)
/* Producer thread -- create new messages at random intervals. */
{
   srand ((DWORD)time(NULL)); /* Seed the random # generator */

   while (!mBlock.fStop) {
      /* Random Delay */
      Sleep(rand()/100);

      /* Get the buffer, fill it */
      EnterCriticalSection (&mBlock.mGuard);
      __try {
         if (!mBlock.fStop) {
            mBlock.fReady = 0;
            MessageFill (&mBlock);
            mBlock.fReady = 1;
            InterlockedIncrement (&mBlock.mSequence);
            }
         }
         __finally { LeaveCriticalSection (&mBlock.mGuard); }
      }
      return 0;
   }

   DWORD WINAPI Consume (void *arg)
   {
      CHAR command, extra;
      /* Consume the NEXT message when prompted by the user */
      while (!mBlock.fStop) { /* Only thread accessing stdin, stdout */
         _tprintf (_T("\n**Enter 'c' for Consume; 's' to stop: "));
         _tscanf ("%c%c", &command, &extra);
         if (command == 's') {
            /* ES not needed here. This is not a read/modify/write.
             * The producer sees the new value after consumer returns */
            mBlock.fStop = 1;
         } else if (command == 'c') { /* Get a new buffer to consume */
            EnterCriticalSection (&mBlock.mGuard);
            __try {
               if (mBlock.fReady == 0)
                  _tprintf (_T("No new messages. Try again later\n"));
               else {
                  MessageDisplay (&mBlock);
                  mBlock.nLost = mBlock.mSequence - mBlock.nCons + 1;
                  mBlock.fReady = 0; /* No new messages are ready */
                  InterlockedIncrement(&mBlock.nCons);
               }
            }
            __finally { LeaveCriticalSection (&mBlock.mGuard); }
         } else {
            _tprintf (_T("Illegal command. Try again.\n"));
         }
      }
      return 0;
   }

   void MessageFill (MSG_BLOCK *msgBlock)
   {
      /* Fill the message buffer, including checksum and timestamp. */
      DWORD i;
      msgBlock->mChecksum = 0;
      for (i = 0; i < DATA_SIZE; i++) {
         msgBlock->mData[i] = rand();
         msgBlock->mChecksum ^= msgBlock->mData[i];
      }
      msgBlock->mTimestamp = time(NULL);
      return;
   }
```