CHAPTER 29

# Primitive Thread Synchronization Constructs

In this chapter:



When a thread pool thread blocks, the thread pool creates additional threads, and the time and memory resources required to create, destroy, and schedule threads is very expensive. When many developers see that they have threads in their program that are not doing anything useful, they tend to create more threads in hopes that the new threads will do something useful. The key to building scalable and responsive applications is to not block the threads you have so that they can be used and reused to execute other tasks. Chapter 27, "Compute-Bound Asynchronous Operations," focused on how to use existing threads to perform compute-bound operations, and Chapter 28, "I/O-Bound Asynchronous Operations," focused on how to use threads when performing I/O-bound operations.

In this chapter, I focus on thread synchronization. Thread synchronization is used to prevent corruption when multiple threads access shared data *at the same time*. I emphasize *at the same time*, because thread synchronization is all about timing. If you have some data that is accessed by two threads and those threads cannot possibly touch the data simultaneously, then thread synchronization is not required at all. In Chapter 28, I discussed how different sections of async functions can be executed by different threads. Here we could potentially have two different threads accessing the same variables and data. But async functions are implemented in such a way that it is impossible for two threads to access this same data at the same time. Therefore, no thread synchronization is required when code accesses data contained within the async function.

This is ideal because thread synchronization has many problems associated with it. First, it is tedious and extremely error-prone. In your code, you must identify all data that could potentially be touched by multiple threads at the same time. Then you must surround this code with additional code that acquires and releases a thread synchronization lock. The lock ensures that only one thread at a time can access the resource. If you forget to surround just one block of code with a lock, then the data will become corrupted. Also, there is no way to prove that you have added all your locking code correctly. You just have to run your application, stress-test it a lot, and hope that nothing goes wrong.

In fact, you should test your application on a machine that has as many CPUs as possible because the more CPUs you have, the better chance that two or more threads will attempt to access the resource at the same time, making it more likely you'll detect a problem.

The second problem with locks is that they hurt performance. It takes time to acquire and release a lock because there are additional method calls, and because the CPUs must coordinate with each other to determine which thread will acquire the lock first. Having the CPUs in the machine communicate with each other this way hurts performance. For example, let's say that you have code that adds a node to the head of a linked list.

```
// This class is used by the LinkedList class
public class Node {
   internal Node m_next;
   // Other members not shown
}

public sealed class LinkedList {
   private Node m_head;

   public void Add(Node newNode) {
      // The following two lines perform very fast reference assignments
      newNode.m_next = m_head;
      m_head = newNode;
   }
}
```

This Add method simply performs two reference assignments that can execute extremely fast. Now, if we want to make Add thread safe so that multiple threads can call it simultaneously without corrupting the linked list, then we need to have the Add method acquire and release a lock.

```
public sealed class LinkedList {
   private SomeKindOfLock m_lock = new SomeKindOfLock();
   private Node m_head;

   public void Add(Node newNode) {
      m_lock.Enter();
      // The following two lines perform very fast reference assignments
      newNode.m_next = m_head;
      m_head = newNode;
      m_lock.Leave();
   }
}
```

Although Add is now thread safe, it has also become substantially slower. How much slower depends on the kind of lock chosen; I will compare the performance of various locks in this chapter and in Chapter 30, "Hybrid Thread Synchronization Constructs." But even the fastest lock could make the Add method several times slower than the version of it that didn't have any lock code in it at all. Of course, the performance becomes significantly worse if the code calls Add in a loop to insert several nodes into the linked list.

The third problem with thread synchronization locks is that they allow only one thread to access the resource at a time. This is the lock's whole reason for existing, but it is also a problem, because blocking a thread causes more threads to be created. So, for example, if a thread pool thread attempts to acquire a lock that it cannot have, it is likely that the thread pool will create a new thread to keep the CPUs saturated with work. As discussed in Chapter 26, "Thread Basics," creating a thread is very expensive in terms of both memory and performance. And to make matters even worse, when the blocked thread gets to run again, it will run with this new thread pool thread; Windows is now scheduling more threads than there are CPUs, and this increases context switching, which also hurts performance.

The summary of all of this is that thread synchronization is bad, so you should try to design your applications to avoid as much of it as possible. To that end, you should avoid shared data such as `static` fields. When a thread uses the `new` operator to construct an object, the `new` operator returns a reference to the `new` object. At this point in time, only the thread that constructs the object has a reference to it; no other thread can access that object. If you avoid passing this reference to another thread that might use the object at the same time as the creating thread, then there is no need to synchronize access to the object.

Try to use value types because they are always copied, so each thread operates on its own copy. Finally, it is OK to have multiple threads accessing shared data simultaneously if that access is read-only. For example, many applications create some data structures during their initialization. Once initialized, the application can create as many threads as it wants; if all these threads just query the data, then all the threads can do this simultaneously without acquiring or releasing any locks. The `String` type is an example of this: after a `String` object is created, it is immutable; so many threads can access a single `String` object at the same time without any chance of the `String` object becoming corrupted.

## Class Libraries and Thread Safety

Now, I'd like to say a quick word about class libraries and thread synchronization. Microsoft's Framework Class Library (FCL) guarantees that all static methods are thread safe. This means that if two threads call a static method at the same time, no data will get corrupted. The FCL had to do this internally because there is no way that multiple companies producing different assemblies could coordinate on a single lock for arbitrating access to the resource. The `Console` class contains a `static` field, inside which many of its methods acquire and release to ensure that only one thread at a time is accessing the console.

For the record, making a method thread safe does not mean that it internally takes a thread synchronization lock. A thread-safe method means that data doesn't get corrupted if two threads attempt to access the data at the same time. The `System.Math` class has a `static Max` method implemented as follows.

```
public static Int32 Max(Int32 val1, Int32 val2) {
   return (val1 < val2) ? val2 : val1;
}
```

This method is thread safe even though it doesn't take any lock. Because `Int32` is a value type, the two `Int32` values passed to `Max` are copied into it and so, multiple threads could be calling `Max` simultaneously, but each thread is working on its own data, isolated from any other thread.

On the other hand, the FCL does not guarantee that instance methods are thread safe because adding all the locking code would hurt performance too much. And, in fact, if every instance method acquires and releases a lock, then you ultimately end up having just one thread running in your application at any given time, which hurts performance even more. As mentioned earlier, when a thread constructs an object, only this thread has a reference to the object, no other thread can access that object, and no thread synchronization is required when invoking instance methods. However, if the thread then exposes the reference to the object—by placing it in a `static` field, passing as the state argument to `ThreadPool.QueueUserWorkItem` or to a `Task`, and so on—then thread synchronization is required if the threads could attempt simultaneous non-read-only access.

It is recommended that your own class libraries follow this pattern; that is, make all your static methods thread safe and make all your instance methods not thread safe. There is one caveat to this pattern: if the purpose of the instance method is to coordinate threads, then the instance method should be thread safe. For example, one thread can cancel an operation by calling `CancellationTokenSource`'s `Cancel` method, and another thread detects that it should stop what it's doing by querying the corresponding `CancellationToken`'s `IsCancellationRequested` property. These two instance members have some special thread synchronization code inside them to ensure that the coordination of the two threads goes as expected.[1]

## Primitive User-Mode and Kernel-Mode Constructs

In this chapter, I explain the *primitive* thread synchronization constructs. By *primitive*, I mean the simplest constructs that are available to use in your code. There are two kinds of primitive constructs: user-mode and kernel-mode. Whenever possible, you should use the primitive user-mode constructs, because they are significantly faster than the kernel-mode constructs because they use special CPU instructions to coordinate threads. This means that the coordination is occurring in hardware (which is what makes it fast). But this also means that the Windows operating system never detects that a thread is blocked on a primitive user-mode construct. Because a thread pool thread blocked on a user-mode primitive construct is never considered blocked, the thread pool will not create a new thread to replace the temporarily blocked thread. In addition, these CPU instructions block the thread for an incredibly short period of time.

Wow! All of this sounds great, doesn't it? And it is great, which is why I recommend using these constructs as much as possible. However, there is a downside—only the Windows operating system kernel can stop a thread from running so that it is not wasting CPU time. A thread running in user mode can be preempted by the system, but the thread will be scheduled again as soon as possible. So, a thread that wants to acquire some resource, but can't get it, spins in user mode. This potentially

[1] Specifically, the field that both members access is marked as `volatile`, a concept that will be discussed later in this chapter.

wastes a lot of CPU time, which would be better spent performing other work or even just letting the CPU go idle to conserve power.

This brings us to the primitive kernel-mode constructs. The kernel-mode constructs are provided by the Windows operating system itself. As such, they require that your application's threads call functions implemented in the operating system kernel. Having threads transition from user mode to kernel mode and back incurs a big performance hit, which is why kernel-mode constructs should be avoided.[2] However, they do have a positive feature—when a thread uses a kernel-mode construct to acquire a resource that another thread has, Windows blocks the thread so that it is no longer wasting CPU time. Then, when the resource becomes available, Windows resumes the thread, allowing it to access the resource.

A thread waiting on a construct might block forever if the thread currently holding the construct never releases it. If the construct is a user-mode construct, the thread is running on a CPU forever, and we call this a *livelock*. If the construct is a kernel-mode construct, the thread is blocked forever, and we call this a *deadlock*. Both of these are bad, but of the two, a deadlock is always preferable to a livelock, because a livelock wastes both CPU time and memory (the thread's stack, etc.), whereas a deadlock wastes only memory.[3]

In an ideal world, we'd like to have constructs that take the best of both worlds. That is, we'd like a construct that is fast and non-blocking (like the user-mode constructs) when there is no contention. But when there is contention for the construct, we'd like it to be blocked by the operating system kernel. Constructs that work like this do exist; I call them *hybrid constructs*, and I will discuss them in Chapter 30. It is very common for applications to use the hybrid constructs, because in most applications, it is rare for two or more threads to attempt to access the same data at the same time. A hybrid construct keeps your application running fast most of the time, and occasionally it runs slowly to block the thread. The slowness usually doesn't matter at this point, because your thread is going to be blocked anyway.

Many of the CLR's thread synchronization constructs are really just object-oriented class wrappers around Win32 thread synchronization constructs. After all, CLR threads are Windows threads, which means that Windows schedules and controls the synchronization of threads. Windows thread synchronization constructs have been around because 1992, and a ton of material has been written about them.[4] Therefore, I give them only cursory treatment in this chapter.

---

[2] I'll show a program that measures the performance later in this chapter, at the end of the "Event Constructs" section.

[3] I say that the memory allocated for the thread is wasted because the memory is not being used in a productive manner if the thread is not making forward progress.

[4] In fact, Christophe Nasarre's and my book, *Windows via C/C++*, Fifth Edition (Microsoft Press, 2007), has several chapters devoted to this subject.

The CLR guarantees that reads and writes to variables of the following data types are atomic: `Boolean`, `Char`, `(S)Byte`, `(U)Int16`, `(U)Int32`, `(U)IntPtr`, `Single`, and reference types. This means that all bytes within that variable are read from or written to all at once. So, for example, if you have the following class:

```
internal static class SomeType {
   public static Int32 x = 0;
}
```

and if a thread executes this line of code:

```
SomeType.x = 0x01234567;
```

then the `x` variable will change from `0x00000000` to `0x01234567` all at once (atomically). Another thread cannot possibly see the value in an intermediate state. For example, it is impossible for some other read to query `SomeType.x` and get a value of `0x01230000`. Suppose that the `x` field in the preceding `SomeType` class is an `Int64`. If a thread executes this line of code:

```
SomeType.x = 0x0123456789abcdef;
```

it is possible that another thread could query `x` and get a value of `0x0123456700000000` or `0x0000000089abcdef`, because the read and write operations are not atomic. This is called a torn read.

Although atomic access to variable guarantees that the read or write happens all at once, it does not guarantee *when* the read or write will happen due to compiler and CPU optimizations. The primitive user-mode constructs discussed in this section are used to enforce the timing of these atomic read and write operations. In addition, these constructs can also force atomic and timed access to variables of additional data types: `(U)Int64` and `Double`.

There are two kinds of primitive user-mode thread synchronization constructs:

- Volatile constructs, which perform an atomic read or write operation on a variable containing a simple data type at a specific time
- Interlocked constructs, which perform an atomic read and write operation on a variable containing a simple data type at a specific time

All the volatile and interlocked constructs require you to pass a reference (memory address) to a variable containing a simple data type.

## Volatile Constructs

Back in the early days of computing, software was written using assembly language. Assembly language is very tedious, because programmers must explicitly state everything—use this CPU register for this, branch to that, call indirect through this other thing, and so on. To simplify programming,

higher-level languages were introduced. These higher-level languages introduced common useful constructs, like `if/else`, `switch/case`, various loops, local variables, arguments, virtual method calls, operator overloads, and much more. Ultimately, these language compilers must convert the high-level constructs down to the low-level constructs so that the computer can actually do what you want it to do.

In other words, the C# compiler translates your C# constructs into Intermediate Language (IL), which is then converted by the just-in-time (JIT) compiler into native CPU instructions, which must then be processed by the CPU itself. In addition, the C# compiler, the JIT compiler, and even the CPU itself can optimize your code. For example, the following ridiculous method can ultimately be compiled into nothing.

```
private static void OptimizedAway() {
   // Constant expression is computed at compile time resulting in zero
   Int32 value = (1 * 100) - (50 * 2);

   // If value is 0, the loop never executes
   for (Int32 x = 0; x < value; x++) {
       // There is no need to compile the code in the loop because it can never execute
      Console.WriteLine("Jeff");
   }
}
```

In this code, the compiler can see that `value` will always be 0; therefore, the loop will never execute and consequently, there is no need to compile the code inside the loop. This method could be compiled down to nothing. In fact, when JITting a method that calls `OptimizedAway`, the JITter will try to inline the `OptimizedAway` method's code. Because there is no code, the JITter will even remove the code that tries to call `OptimizedAway`. We love this feature of compilers. As developers, we get to write the code in the way that makes the most sense to us. The code should be easy to write, read, and maintain. Then compilers translate our intentions into machine-understandable code. We want our compilers to do the best job possible for us.

When the C# compiler, JIT compiler, and CPU optimize our code, they guarantee us that the intention of the code is preserved. That is, from a single-threaded perspective, the method does what we want it to do, although it may not do it exactly the way we described in our source code. However, the intention might not be preserved from a multithreaded perspective. Here is an example where the optimizations make the program not work as expected.

```
internal static class StrangeBehavior {
   // As you'll see later, mark this field as volatile to fix the problem
   private static Boolean s_stopWorker = false;

   public static void Main() {
      Console.WriteLine("Main: letting worker run for 5 seconds");
      Thread t = new Thread(Worker);
      t.Start();
      Thread.Sleep(5000);
      s_stopWorker = true;
      Console.WriteLine("Main: waiting for worker to stop");
      t.Join();
   }
```

```
   private static void Worker(Object o) {
      Int32 x = 0;
      while (!s_stopWorker) x++;
      Console.WriteLine("Worker: stopped when x={0}", x);
   }
}
```

In this code, the `Main` method creates a new thread that executes the `Worker` method. This `Worker` method counts as high as it can before being told to stop. The `Main` method allows the `Worker` thread to run for five seconds before telling it to stop by setting the `static Boolean` field to `true`. At this point, the `Worker` thread should display what it counted up to, and then the thread will terminate. The `Main` thread waits for the `Worker` thread to terminate by calling `Join`, and then the `Main` thread returns, causing the whole process to terminate.

Looks simple enough, right? Well, the program has a potential problem due to all the optimizations that could happen to it. You see, when the `Worker` method is compiled, the compiler sees that `s_stopWorker` is either `true` or `false`, and it also sees that this value never changes inside the `Worker` method itself. So the compiler could produce code that checks `s_stopWorker` first. If `s_stopWorker` is `true`, then `Worker: stopped when x=0` will be displayed. If `s_stopWorker` is `false`, then the compiler produces code that enters an infinite loop that increments `x` forever. You see, the optimizations cause the loop to run very fast because checking `s_stopWorker` only occurs once before the loop; it does not get checked with each iteration of the loop.

If you actually want to see this in action, put this code in a .cs file and compile the code by using C#'s `/platform:x86` and `/optimize+` switches. Then run the resulting EXE file, and you'll see that the program runs forever. Note that you have to compile for x86, ensuring that the x86 JIT compiler is used at run time. The x86 JIT compiler is more mature than the x64 JIT compiler, so it performs more aggressive optimizations. The x64 JIT compiler does not perform this particular optimization, and therefore the program runs to completion. This highlights another interesting point about all of this. Whether your program behaves as expected depends on a lot of factors, such as which compiler version and compiler switches are used, which JIT compiler is used, and which CPU your code is running on. In addition, to see the preceding program run forever, you must not run the program under a debugger because the debugger causes the JIT compiler to produce unoptimized code that is easier to step through.

Let's look at another example, which has two threads that are both accessing two fields.

```
internal sealed class ThreadsSharingData {
   private Int32 m_flag = 0;
   private Int32 m_value = 0;

   // This method is executed by one thread
   public void Thread1() {
      // Note: These could execute in reverse order
      m_value = 5;
      m_flag  = 1;
   }

   // This method is executed by another thread
   public void Thread2() {
```

```
      // Note: m_value could be read before m_flag
      if (m_flag == 1)
         Console.WriteLine(m_value);
   }
}
```

The problem with this code is that the compilers/CPU could translate the code in such a way as to reverse the two lines of code in the `Thread1` method. After all, reversing the two lines of code does not change the intention of the method. The method needs to get a 5 in `m_value` and a 1 in `m_flag`. From a single-threaded application's perspective, the order of executing this code is unimportant. If these two lines do execute in reverse order, then another thread executing the `Thread2` method could see that `m_flag` is 1 and then display 0.

Let's look at this code another way. Let's say that the code in the `Thread1` method executes in *program order* (the way it was written). When compiling the code in the `Thread2` method, the compiler must generate code to read `m_flag` and `m_value` from RAM into CPU registers. It is possible that RAM will deliver the value of `m_value` first, which would contain a 0. Then the `Thread1` method could execute, changing `m_value` to 5 and `m_flag` to 1. But `Thread2`'s CPU register doesn't see that `m_value` has been changed to 5 by this other thread, and then the value in `m_flag` could be read from RAM into a CPU register and the value of `m_flag` becomes 1 now, causing `Thread2` to again display 0.

This is all very scary stuff and is more likely to cause problems in a release build of your program than in a debug build of your program, making it particularly tricky to detect these problems and correct your code. Now, let's talk about how to correct your code.

The static `System.Threading.Volatile` class offers two static methods that look like this.[5]

```
public static class Volatile {
   public static void  Write(ref Int32 location, Int32 value);
   public static Int32 Read(ref Int32 location);
}
```

These methods are special. In effect, these methods disable some optimizations usually performed by the C# compiler, the JIT compiler, and the CPU itself. Here's how the methods work:

- The `Volatile.Write` method forces the value in `location` to be written to at the point of the call. In addition, any *earlier* program-order loads and stores must occur *before* the call to `Volatile.Write`.
- The `Volatile.Read` method forces the value in `location` to be read from at the point of the call. In addition, any *later* program-order loads and stores must occur *after* the call to `Volatile.Read`.

[5] There are also overloads of `Read` and `Write` that operate on the following types: `Boolean`, `(S)Byte`, `(U)Int16`, `UInt32`, `(U)Int64`, `(U)IntPtr`, `Single`, `Double`, and `T` where `T` is a generic type constrained to '`class`' (reference types).

**Important** I know that this can be very confusing, so let me summarize it as a simple rule. When threads are communicating with each other via shared memory, write the last value by calling `Volatile.Write` and read the first value by calling `Volatile.Read`.

So now we can fix the `ThreadsSharingData` class by using these methods.

```
internal sealed class ThreadsSharingData {
    private Int32 m_flag = 0;
    private Int32 m_value = 0;

    // This method is executed by one thread
    public void Thread1() {
        // Note: 5 must be written to m_value before 1 is written to m_flag
        m_value = 5;
        Volatile.Write(ref m_flag, 1);
    }

    // This method is executed by another thread
    public void Thread2() {
        // Note: m_value must be read after m_flag is read
        if (Volatile.Read(ref m_flag) == 1)
            Console.WriteLine(m_value);
    }
}
```

First, notice that we are following the rule. The `Thread1` method writes two values out to fields that are shared by multiple threads. The last value that we want written (setting `m_flag` to `1`) is performed by calling `Volatile.Write`. The `Thread2` method reads two values from fields shared by multiple threads, and the first value being read (`m_flag`) is performed by calling `Volatile.Read`.

But what is really happening here? Well, for the `Thread1` method, the `Volatile.Write` call ensures that all the writes above it are completed before a `1` is written to `m_flag`. Because `m_value = 5` is before the call to `Volatile.Write`, it must complete first. In fact, if there were many variables being modified before the call to `Volatile.Write`, they would all have to complete before `1` is written to `m_flag`. Note that the writes before the call to `Volatile.Write` can be optimized to execute in any order; it's just that all the writes have to complete before the call to `Volatile.Write`.

For the `Thread2` method, the `Volatile.Read` call ensures that all variable reads after it start after the value in `m_flag` has been read. Because reading `m_value` is after the call to `Volatile.Read`, the value must be read after having read the value in `m_flag`. If there were many reads after the call to `Volatile.Read`, they would all have to start after the value in `m_flag` has been read. Note that the reads after the call to `Volatile.Read` can be optimized to execute in any order; it's just that the reads can't start happening until after the call to `Volatile.Read`.

## C#'s Support for Volatile Fields

Making sure that programmers call the `Volatile.Read` and `Volatile.Write` methods correctly is a lot to ask. It's hard for programmers to keep all of this in their minds and to start imagining what other threads might be doing to shared data in the background. To simplify this, the C# compiler has the `volatile` keyword, which can be applied to static or instance fields of any of these types: `Boolean`, `(S)Byte`, `(U)Int16`, `(U)Int32`, `(U)IntPtr`, `Single`, or `Char`. You can also apply the `volatile` keyword to reference types and any enum field as long as the enumerated type has an underlying type of `(S)Byte`, `(U)Int16`, or `(U)Int32`. The JIT compiler ensures that all accesses to a volatile field are performed as volatile reads and writes, so that it is not necessary to explicitly call `Volatile`'s static `Read` or `Write` methods. Furthermore, the `volatile` keyword tells the C# and JIT compilers not to cache the field in a CPU register, ensuring that all reads to and from the field actually cause the value to be read from memory.

Using the `volatile` keyword, we can rewrite the `ThreadsSharingData` class as follows.

```
internal sealed class ThreadsSharingData {
   private volatile Int32 m_flag = 0;
   private          Int32 m_value = 0;

   // This method is executed by one thread
   public void Thread1() {
      // Note: 5 must be written to m_value before 1 is written to m_flag
      m_value = 5;
      m_flag = 1;
   }

   // This method is executed by another thread
   public void Thread2() {
      // Note: m_value must be read after m_flag is read
      if (m_flag == 1)
         Console.WriteLine(m_value);
   }
}
```

There are some developers (and I am one of them) who do not like C#'s `volatile` keyword, and they think that the language should not provide it.[6] Our thinking is that most algorithms require few volatile `read` or `write` accesses to a field and that most other accesses to the field can occur normally, improving performance; seldom is it required that all accesses to a field be volatile. For example, it is difficult to interpret how to apply volatile read operations to algorithms like this one.

```
m_amount = m_amount + m_amount; // Assume m_amount is a volatile field defined in a class
```

Normally, an integer number can be doubled simply by shifting all bits left by 1 bit, and many compilers can examine the preceding code and perform this optimization. However, if `m_amount` is a `volatile` field, then this optimization is not allowed. The compiler must produce code to read `m_amount` into a register and then read it again into another register, add the two registers together,

[6] By the way, it is good to see that Microsoft Visual Basic does not offer a volatile semantic built into its language.

and then write the result back out to the `m_amount` field. The unoptimized code is certainly bigger and slower; it would be unfortunate if it were contained inside a loop.

Furthermore, C# does not support passing a `volatile` field by reference to a method. For example, if `m_amount` is defined as a `volatile Int32`, attempting to call `Int32`'s `TryParse` method causes the compiler to generate a warning as shown here.

```
Boolean success = Int32.TryParse("123", out m_amount);
// The preceding line causes the C# compiler to generate a warning:
// CS0420: a reference to a volatile field will not be treated as volatile
```

Finally, volatile fields are not Common Language Specification (CLS) compliant because many languages (including Visual Basic) do not support them.

## Interlocked Constructs

`Volatile`'s `Read` method performs an atomic read operation, and its `Write` method performs an atomic write operation. That is, each method performs either an atomic read operation *or* an atomic write operation. In this section, we look at the static `System.Threading.Interlocked` class's methods. Each of the methods in the `Interlocked` class performs an atomic read *and* write operation. In addition, all the `Interlocked` methods are full memory fences. That is, any variable writes before the call to an `Interlocked` method execute before the `Interlocked` method, and any variable reads after the call execute after the call.

The static methods that operate on `Int32` variables are by far the most commonly used methods. I show them here.

```
public static class Interlocked {
   // return (++location)
   public static Int32 Increment(ref Int32 location);

   // return (--location)
   public static Int32 Decrement(ref Int32 location);

   // return (location += value)
   // Note: value can be a negative number allowing subtraction
   public static Int32 Add(ref Int32 location, Int32 value);

   // Int32 old = location; location = value; return old;
   public static Int32 Exchange(ref Int32 location, Int32 value);

   // Int32 old = location;
   // if (location == comparand) location = value;
   // return old;
   public static Int32 CompareExchange(ref Int32 location, Int32 value, Int32 comparand);
   ...
}
```

There are also overloads of the preceding methods that operate on `Int64` values. Furthermore, the `Interlocked` class offers `Exchange` and `CompareExchange` methods that take `Object`, `IntPtr`, `Single`, and `Double`, and there is also a generic version in which the generic type is constrained to `class` (any reference type).

Personally, I love the `Interlocked` methods, because they are relatively fast and you can do so much with them. Let me show you some code that uses the `Interlocked` methods to asynchronously query several web servers and concurrently process the returned data. This code is pretty short, never blocks any threads, and uses thread pool threads to scale automatically, consuming up to the number of CPUs available if its workload could benefit from it. In addition, the code, as is, supports accessing up to 2,147,483,647 (`Int32.MaxValue`) web servers. In other words, this code is a great model to follow for your own scenarios.

```
internal sealed class MultiWebRequests {
   // This helper class coordinates all the asynchronous operations
   private AsyncCoordinator m_ac = new AsyncCoordinator();

   // Set of web servers we want to query & their responses (Exception or Int32)
   // NOTE: Even though multiple could access this dictionary simultaneously,
   // there is no need to synchronize access to it because the keys are
   // read-only after construction
   private Dictionary<String, Object> m_servers = new Dictionary<String, Object> {
      { "http://Wintellect.com/", null },
      { "http://Microsoft.com/",  null },
      { "http://1.1.1.1/",        null }
   };

   public MultiWebRequests(Int32 timeout = Timeout.Infinite) {
      // Asynchronously initiate all the requests all at once
      var httpClient = new HttpClient();
      foreach (var server in m_servers.Keys) {
         m_ac.AboutToBegin(1);
         httpClient.GetByteArrayAsync(server)
            .ContinueWith(task => ComputeResult(server, task));
      }

      // Tell AsyncCoordinator that all operations have been initiated and to call
      // AllDone when all operations complete, Cancel is called, or the timeout occurs
      m_ac.AllBegun(AllDone, timeout);
   }

   private void ComputeResult(String server, Task<Byte[]> task) {
      Object result;
      if (task.Exception != null) {
         result = task.Exception.InnerException;
      } else {
         // Process I/O completion here on thread pool thread(s)
         // Put your own compute-intensive algorithm here...
         result = task.Result.Length;   // This example just returns the length
      }
```

```
      // Save result (exception/sum) and indicate that 1 operation completed
      m_servers[server] = result;
      m_ac.JustEnded();
   }

   // Calling this method indicates that the results don't matter anymore
   public void Cancel() { m_ac.Cancel(); }

   // This method is called after all web servers respond,
   // Cancel is called, or the timeout occurs
   private void AllDone(CoordinationStatus status) {
      switch (status) {
         case CoordinationStatus.Cancel:
            Console.WriteLine("Operation canceled.");
            break;

         case CoordinationStatus.Timeout:
            Console.WriteLine("Operation timed-out.");
            break;

         case CoordinationStatus.AllDone:
            Console.WriteLine("Operation completed; results below:");
            foreach (var server in m_servers) {
               Console.Write("{0} ", server.Key);
               Object result = server.Value;
               if (result is Exception) {
                  Console.WriteLine("failed due to {0}.", result.GetType().Name);
               } else {
                  Console.WriteLine("returned {0:N0} bytes.", result);
               }
            }
            break;
      }
   }
}
```

OK, the preceding code doesn't actually use any `Interlocked` methods directly, because I encapsulated all the coordination code in a reusable class called `AsyncCoordinator`, which I'll explain shortly. Let me first explain what this class is doing. When the `MultiWebRequest` class is constructed, it initializes an `AsyncCoordinator` and a dictionary containing the set of server URIs (and their future result). It then issues all the web requests asynchronously one right after the other. It does this by first calling `AsyncCoordinator`'s `AboutToBegin` method, passing it the number of requests about to be issued.[7] Then it initiates the request by calling `HttpClient`'s `GetByteArrayAsync`. This returns a `Task` and I then call `ContinueWith` on this `Task` so that when the server replies with the bytes, they can be processed by my `ComputeResult` method concurrently via many thread pool threads. After all the web servers' requests have been made, the `AsyncCoordinator`'s `AllBegun` method is called, passing it the name of the method (`AllDone`) that should execute when all the operations complete and a timeout value. As each web server responds, various thread pool threads

[7] The code would still work correctly if it was rewritten calling `m_ac.AboutToBegin(m_requests.Count)` just once before the `for` loop instead of calling `AboutToBegin` inside the loop.

will call the MultiWebRequests's ComputeResult method. This method processes the bytes returned from the server (or any error that may have occurred) and saves the result in the dictionary collection. After storing each result, AsyncCoordinator's JustEnded method is called to let the AsyncCoordinator object know that an operation completed.

If all the operations have completed, then the AsyncCoordinator will invoke the AllDone method to process the results from all the web servers. The code executing the AllDone method will be the thread pool thread that just happened to get the last web server response. If timeout or cancellation occurs, then AllDone will be invoked via whatever thread pool thread notifies the AsyncCoordinator of timeout or using whatever thread happened to call the Cancel method. There is also a chance that the thread issuing the web server requests could invoke AllDone itself if the last request completes before AllBegun is called.

Note that there is a race because it is possible that all web server requests complete, AllBegun is called, timeout occurs, and Cancel is called all at the exact same time. If this happens, then the AsyncCoordinator will select a winner and three losers, ensuring that the AllDone method is never called more than once. The winner is identified by the status argument passed into AllDone, which can be one of the symbols defined by the CoordinationStatus type.

```
internal enum CoordinationStatus { AllDone, Timeout, Cancel };
```

Now that you get a sense of what happens, let's take a look at how it works. The AsyncCoordinator class encapsulates all the thread coordination logic in it. It uses Interlocked methods for everything to ensure that the code runs extremely fast and that no threads ever block. Here is the code for this class.

```
internal sealed class AsyncCoordinator {
   private Int32 m_opCount = 1;        // Decremented when AllBegun calls JustEnded
   private Int32 m_statusReported = 0; // 0=false, 1=true
   private Action<CoordinationStatus> m_callback;
   private Timer m_timer;

   // This method MUST be called BEFORE initiating an operation
   public void AboutToBegin(Int32 opsToAdd = 1) {
      Interlocked.Add(ref m_opCount, opsToAdd);
   }

   // This method MUST be called AFTER an operation's result has been processed
   public void JustEnded() {
      if (Interlocked.Decrement(ref m_opCount) == 0)
         ReportStatus(CoordinationStatus.AllDone);
   }

   // This method MUST be called AFTER initiating ALL operations
   public void AllBegun(Action<CoordinationStatus> callback,
      Int32 timeout = Timeout.Infinite) {
```

```
      m_callback = callback;
      if (timeout != Timeout.Infinite)
         m_timer = new Timer(TimeExpired, null, timeout, Timeout.Infinite);
      JustEnded();
   }

   private void TimeExpired(Object o) { ReportStatus(CoordinationStatus.Timeout); }
   public void Cancel()               { ReportStatus(CoordinationStatus.Cancel); }

   private void ReportStatus(CoordinationStatus status) {
      // If status has never been reported, report it; else ignore it
      if (Interlocked.Exchange(ref m_statusReported, 1) == 0)
         m_callback(status);
   }
}
```

The most important field in this class is the `m_opCount` field. This field keeps track of the number of asynchronous operations that are still outstanding. Just before each asynchronous operation is started, `AboutToBegin` is called. This method calls `Interlocked.Add` to add the number passed to it to the `m_opCount` field in an atomic way. Adding to `m_opCount` must be performed atomically because web servers could be processing responses on thread pool threads as more operations are being started. As web server responses are processed, `JustEnded` is called. This method calls `Interlocked.Decrement` to atomically subtract 1 from `m_opCount`. Whichever thread happens to set `m_opCount` to 0 calls `ReportStatus`.

**Note** The `m_opCount` field is initialized to 1 (not 0); this is critically important because it ensures that `AllDone` is not invoked while the thread executing the constructor method is still issuing web server requests. Before the constructor calls `AllBegun`, there is no way that `m_opCount` will ever reach 0. When the constructor calls `AllBegun`, `AllBegun` internally calls `JustEnded`, which decrements `m_opCount` and effectively undoes the effect of having initialized it to 1. Now, `m_opCount` can reach 0, but only after we know that all the web server requests have been initiated.

The `ReportStatus` method arbitrates the race that can occur among all the operations completing, the timeout occurring, and `Cancel` being called. `ReportStatus` must make sure that only one of these conditions is considered the winner so that the `m_callback` method is invoked only once. Arbitrating the winner is done via calling `Interlocked.Exchange`, passing it a reference to the `m_statusReported` field. This field is really treated as a `Boolean` variable; however, it can't actually be a `Boolean` variable because there are no `Interlocked` methods that accept a `Boolean` variable. So I use an `Int32` variable instead where 0 means `false` and 1 means `true`.

Inside `ReportStatus`, the `Interlocked.Exchange` call will change `m_statusReported` to 1. But only the first thread to do this will see `Interlocked.Exchange` return a 0, and only this thread will invoke the callback method. Any other threads that call `Interlocked.Exchange` will get a return value of 1, effectively notifying these threads that the callback method has already been invoked and therefore it should not be invoked again.

## Implementing a Simple Spin Lock

The `Interlocked` methods are great, but they mostly operate on `Int32` values. What if you need to manipulate a bunch of fields in a class object atomically? In this case, we need a way to stop all threads but one from entering the region of code that manipulates the fields. Using `Interlocked` methods, we can build a thread synchronization lock.

```
internal struct SimpleSpinLock {
   private Int32 m_ResourceInUse; // 0=false (default), 1=true

   public void Enter() {
      while (true) {
         // Always set resource to in-use
         // When this thread changes it from not in-use, return
         if (Interlocked.Exchange(ref m_ResourceInUse, 1) == 0) return;
         // Black magic goes here...
      }
   }

   public void Leave() {
      // Set resource to not in-use
      Volatile.Write(ref m_ResourceInUse, 0);
   }
}
```

And here is a class that shows how to use the `SimpleSpinLock`.

```
public sealed class SomeResource {
   private SimpleSpinLock m_sl = new SimpleSpinLock();

   public void AccessResource() {
      m_sl.Enter();
      // Only one thread at a time can get in here to access the resource...
      m_sl.Leave();
   }
}
```

The `SimpleSpinLock` implementation is very simple. If two threads call `Enter` at the same time, `Interlocked.Exchange` ensures that one thread changes `m_resourceInUse` from 0 to 1 and sees that `m_resourceInUse` was 0. This thread then returns from `Enter` so that it can continue executing the code in the `AccessResource` method. The other thread will change `m_resourceInUse` from a 1 to a 1. This thread will see that it did not change `m_resourceInUse` from a 0, and this thread will now start spinning continuously, calling `Exchange` until the first thread calls `Leave`.

When the first thread is done manipulating the fields of the `SomeResource` object, it calls `Leave`, which internally calls `Volatile.Write` and changes `m_resourceInUse` back to a 0. This causes the spinning thread to then change `m_resourceInUse` from a 0 to a 1, and this thread now gets to return from `Enter` so that it can access `SomeResource` object's fields.

There you have it. This is a simple implementation of a thread synchronization lock. The big potential problem with this lock is that it causes threads to spin when there is contention for the lock.

This spinning wastes precious CPU time, preventing the CPU from doing other, more useful work. As a result, spin locks should only ever be used to guard regions of code that execute very quickly.

Spin locks should not typically be used on single-CPU machines, because the thread that holds the lock can't quickly release it if the thread that wants the lock is spinning. The situation becomes much worse if the thread holding the lock is at a lower priority than the thread wanting to get the lock, because now the thread holding the lock may not get a chance to run at all, resulting in a livelock situation. Windows sometimes boosts a thread's priority dynamically for short periods of time. Therefore, boosting should be disabled for threads that are using spin locks; see the `PriorityBoostEnabled` properties of `System.Diagnostics.Process` and `System.Diagnostics.ProcessThread`. There are issues related to using spin locks on hyperthreaded machines, too. In an attempt to circumvent these kinds of problems, many spin locks have some additional logic in them; I refer to the additional logic as *Black Magic*. I'd rather not go into the details of Black Magic because it changes over time as more people study locks and their performance. However, I will say this: The FCL ships with a structure, `System.Threading.SpinWait`, which encapsulates the state-of-the-art thinking around this Black Magic.

### Putting a Delay in the Thread's Processing

The Black Magic is all about having a thread that wants a resource to pause its execution temporarily so that the thread that currently has the resource can execute its code and relinquish the resource. To do this, the `SpinWait struct` internally calls `Thread`'s static `Sleep`, `Yield`, and `SpinWait` methods. I'll briefly describe these methods in this sidebar.

A thread can tell the system that it does not want to be schedulable for a certain amount of time. This is accomplished by calling `Thread`'s static `Sleep` method.

```
public static void Sleep(Int32 millisecondsTimeout);
public static void Sleep(TimeSpan timeout);
```

This method causes the thread to suspend itself until the specified amount of time has elapsed. Calling `Sleep` allows the thread to voluntarily give up the remainder of its time-slice. The system makes the thread not schedulable for *approximately* the amount of time specified. That's right—if you tell the system you want a thread to sleep for 100 milliseconds, the thread will sleep approximately that long, but possibly several seconds or even minutes more. Remember that Windows is not a real-time operating system. Your thread will probably wake up at the right time, but whether it does depends on what else is going on in the system.

You can call `Sleep` and pass the value in `System.Threading.Timeout.Infinite` (defined as –1) for the `millisecondsTimeout` parameter. This tells the system to never schedule the thread, and it is not a useful thing to do. It is much better to have the thread exit and then recover its stack and kernel object. You can pass 0 to `Sleep`. This tells the system that the calling thread relinquishes the remainder of its current time-slice, and it forces the system to schedule another thread. However, the system can reschedule the thread that just called `Sleep`. This will happen if there are no more schedulable threads at the same priority or higher.

A thread can ask Windows to schedule another thread on the current CPU by calling Thread's Yield method.

```
public static Boolean Yield();
```

If Windows has another thread ready to run on the current processor, then Yield returns true and the thread that called Yield ended its time-slice early, the selected thread gets to run for one time-slice, and then the thread that called Yield is scheduled again and starts running with a fresh new time-slice. If Windows does not have another thread to run on the current processor, then Yield returns false and the thread continues its time-slice.

The Yield method exists in order to give a thread of equal or lower priority that is starving for CPU time a chance to run. A thread calls this method if it wants a resource that is currently owned by another thread. The hope is that Windows will schedule the thread that currently owns the resource and that this thread will relinquish the resource. Then, when the thread that called Yield runs again, this thread can have the resource.

Yield is a cross between calling Thread.Sleep(0) and Thread.Sleep(1). Thread.Sleep(0) will not let a lower-priority thread run, whereas Thread.Sleep(1) will always force a context switch and Windows will force the thread to sleep longer than one millisecond due to the resolution of the internal system timer.

Hyperthreaded CPUs really let only one thread run at a time. So, when executing spin loops on these CPUs, you need to force the current thread to pause so that the CPU switches to the other thread, allowing it to run. A thread can force itself to pause, allowing a hyperthreaded CPU to switch to its other thread by calling Thread's SpinWait method.

```
public static void SpinWait(Int32 iterations);
```

Calling this method actually executes a special CPU instruction; it does not tell Windows to do anything (because Windows already thinks that it has scheduled two threads on the CPU). On a non-hyperthreaded CPU, this special CPU instruction is simply ignored.

**Note** For more information about these methods, see their Win32 equivalents: Sleep, SwitchToThread, and YieldProcessor. You can also learn more about adjusting the resolution of the system timer by looking up the Win32 timeBeginPeriod and timeEndPeriod functions.

The FCL also includes a System.Threading.SpinLock structure that is similar to my SimpleSpinLock class shown earlier, except that it uses the SpinWait structure to improve performance. The SpinLock structure also offers timeout support. By the way, it is interesting to note that my SimpleSpinLock and the FCL's SpinLock are both value types. This means that they are lightweight, memory-friendly objects. A SpinLock is a good choice if you need to associate a lock with each item in a collection, for example. However, you must make sure that you do not pass SpinLock instances

around, because they are copied and you will lose any and all synchronization. And although you can define instance SpinLock fields, do not mark the field as readonly, because its internal state must change as the lock is manipulated.

## The Interlocked Anything Pattern

Many people look at the Interlocked methods and wonder why Microsoft doesn't create a richer set of interlocked methods that can be used in a wider range of scenarios. For example, it would be nice if the Interlocked class offered Multiply, Divide, Minimum, Maximum, And, Or, Xor, and a bunch of other methods. Although the Interlocked class doesn't offer these methods, there is a well-known pattern that allows you to perform any operation on an Int32 in an atomic way by using Interlocked.CompareExchange. In fact, because Interlocked.CompareExchange has additional overloads that operate on Int64, Single, Double, Object, and a generic reference type, this pattern will actually work for all these types, too.

This pattern is similar to optimistic concurrency patterns used for modifying database records. Here is an example of the pattern that is being used to create an atomic Maximum method.

```
public static Int32 Maximum(ref Int32 target, Int32 value) {
   Int32 currentVal = target, startVal, desiredVal;

   // Don't access target in the loop except in an attempt
   // to change it because another thread may be touching it
   do {
      // Record this iteration's starting value
      startVal = currentVal;

      // Calculate the desired value in terms of startVal and value
      desiredVal = Math.Max(startVal, value);

      // NOTE: the thread could be preempted here!

      // if (target == startVal) target = desiredVal
      // Value prior to potential change is returned
      currentVal = Interlocked.CompareExchange(ref target, desiredVal, startVal);

      // If the starting value changed during this iteration, repeat
   } while (startVal != currentVal);

   // Return the maximum value when this thread tried to set it
   return desiredVal;
}
```

Now let me explain exactly what is going on here. Upon entering the method, `currentVal` is initialized to the value in `target` at the moment the method starts executing. Then, inside the loop, `startVal` is initialized to this same value. Using `startVal`, you can perform any operation you want. This operation can be extremely complex, consisting of thousands of lines of code. But, ultimately, you must end up with a result that is placed into `desiredVal`. In my example, I simply determine whether `startVal` or `value` contains the larger value.

Now, while this operation is running, another thread could change the value in `target`. It is unlikely that this will happen, but it is possible. If this does happen, then the value in `desiredVal` is based off an old value in `startVal`, not the current value in `target`, and therefore, we should not change the value in `target`. To ensure that the value in `target` is changed to `desiredVal` if no thread has changed `target` behind our thread's back, we use `Interlocked.CompareExchange`. This method checks whether the value in `target` matches the value in `startVal` (which identifies the value that we thought was in `target` before starting to perform the operation). If the value in `target` didn't change, then `CompareExchange` changes it to the new value in `desiredVal`. If the value in `target` did change, then `CompareExchange` does not alter the value in `target` at all.

`CompareExchange` returns the value that is in `target` at the time when `CompareExchange` is called, which I then place in `currentVal`. Then, a check is made comparing `startVal` with the new value in `currentVal`. If these values are the same, then a thread did not change `target` behind our thread's back, `target` now contains the value in `desiredVal`, the `while` loop does not loop around, and the method returns. If `startVal` is not equal to `currentVal`, then a thread did change the value in `target` behind our thread's back, `target` did not get changed to our value in `desiredVal`, and the `while` loop will loop around and try the operation again, this time using the new value in `currentVal` that reflects the other thread's change.

Personally, I have used this pattern in a lot of my own code and, in fact, I made a generic method, `Morph`, which encapsulates this pattern.[8]

```
delegate Int32 Morpher<TResult, TArgument>(Int32 startValue, TArgument argument,
   out TResult morphResult);

static TResult Morph<TResult, TArgument>(ref Int32 target, TArgument argument,
   Morpher<TResult, TArgument> morpher) {

   TResult morphResult;
   Int32 currentVal = target, startVal, desiredVal;
   do {
      startVal = currentVal;
      desiredVal = morpher(startVal, argument, out morphResult);
      currentVal = Interlocked.CompareExchange(ref target, desiredVal, startVal);
   } while (startVal != currentVal);
   return morphResult;
}
```

[8] Obviously, the `Morph` method incurs a performance penalty due to invoking the `morpher` callback method. For best performance, execute the operation inline, as in the `Maximum` example.

## Kernel-Mode Constructs

Windows offers several kernel-mode constructs for synchronizing threads. The kernel-mode constructs are much slower than the user-mode constructs. This is because they require coordination from the Windows operating system itself. Also, each method call on a kernel object causes the calling thread to transition from managed code to native user-mode code to native kernel-mode code and then return all the way back. These transitions require a lot of CPU time and, if performed frequently, can adversely affect the overall performance of your application.

However, the kernel-mode constructs offer some benefits over the primitive user-mode constructs, such as:

- When a kernel-mode construct detects contention on a resource, Windows blocks the losing thread so that it is not spinning on a CPU, wasting processor resources.
- Kernel-mode constructs can synchronize native and managed threads with each other.
- Kernel-mode constructs can synchronize threads running in different processes on the same machine.
- Kernel-mode constructs can have security applied to them to prevent unauthorized accounts from accessing them.
- A thread can block until all kernel-mode constructs in a set are available or until any one kernel-mode construct in a set has become available.
- A thread can block on a kernel-mode construct specifying a timeout value; if the thread can't have access to the resource it wants in the specified amount of time, then the thread is unblocked and can perform other tasks.

The two primitive kernel-mode thread synchronization constructs are events and semaphores. Other kernel-mode constructs, such as mutex, are built on top of the two primitive constructs. For more information about the Windows kernel-mode constructs, see the book, *Windows via C/C++*, Fifth Edition (Microsoft Press, 2007) by myself and Christophe Nasarre.

The `System.Threading` namespace offers an abstract base class called `WaitHandle`. The `WaitHandle` class is a simple class whose sole purpose is to wrap a Windows kernel object handle. The FCL provides several classes derived from `WaitHandle`. All classes are defined in the `System.Threading` namespace. The class hierarchy looks like this.

```
WaitHandle
   EventWaitHandle
      AutoResetEvent
      ManualResetEvent
   Semaphore
   Mutex
```

Internally, the WaitHandle base class has a SafeWaitHandle field that holds a Win32 kernel object handle. This field is initialized when a concrete WaitHandle-derived class is constructed. In addition, the WaitHandle class publicly exposes methods that are inherited by all the derived classes. Every method called on a kernel-mode construct represents a full memory fence. WaitHandle's interesting public methods are shown in the following code (some overloads for some methods are not shown).

```
public abstract class WaitHandle : MarshalByRefObject, IDisposable {
   // WaitOne internally calls the Win32 WaitForSingleObjectEx function.
   public virtual Boolean WaitOne();
   public virtual Boolean WaitOne(Int32 millisecondsTimeout);
   public virtual Boolean WaitOne(TimeSpan timeout);

   // WaitAll internally calls the Win32 WaitForMultipleObjectsEx function
   public static Boolean WaitAll(WaitHandle[] waitHandles);
   public static Boolean WaitAll(WaitHandle[] waitHandles, Int32 millisecondsTimeout);
   public static Boolean WaitAll(WaitHandle[] waitHandles, TimeSpan timeout);

   // WaitAny internally calls the Win32 WaitForMultipleObjectsEx function
   public static Int32 WaitAny(WaitHandle[] waitHandles);
   public static Int32 WaitAny(WaitHandle[] waitHandles, Int32 millisecondsTimeout);   public
static Int32 WaitAny(WaitHandle[] waitHandles, TimeSpan timeout);
   public const Int32 WaitTimeout = 258; // Returned from WaitAny if a timeout occurs

   //  Dispose internally calls the Win32 CloseHandle function - DON'T CALL THIS.
   public void Dispose();

}
```

There are a few things to note about these methods:

- You call WaitHandle's WaitOne method to have the calling thread wait for the underlying kernel object to become signaled. Internally, this method calls the Win32 WaitForSingleObjectEx function. The returned Boolean is true if the object became signaled or false if a timeout occurs.
- You call WaitHandle's static WaitAll method to have the calling thread wait for all the kernel objects specified in the WaitHandle[] to become signaled. The returned Boolean is true if all of the objects became signaled or false if a timeout occurs. Internally, this method calls the Win32 WaitForMultipleObjectsEx function, passing TRUE for the bWaitAll parameter.
- You call WaitHandle's static WaitAny method to have the calling thread wait for any one of the kernel objects specified in the WaitHandle[] to become signaled. The returned Int32 is the index of the array element corresponding to the kernel object that became signaled, or WaitHandle.WaitTimeout if no object became signaled while waiting. Internally, this method calls the Win32 WaitForMultipleObjectsEx function, passing FALSE for the bWaitAll parameter.

- The array that you pass to the `WaitAny` and `WaitAll` methods must contain no more than 64 elements or else the methods throw a `System.NotSupportedException`.
- You call `WaitHandle`'s `Dispose` method to close the underlying kernel object handle. Internally, these methods call the Win32 `CloseHandle` function. You can only call `Dispose` explicitly in your code if you know for a fact that no other threads are using the kernel object. This puts a lot of burden on you as you write your code and test it. So, I would strongly discourage you from calling `Dispose`; instead, just let the garbage collector (GC) do the cleanup. The GC knows when no threads are using the object anymore, and then it will get rid of it. In a way, the GC is doing thread synchronization for you automatically!

**Note** In some cases, when a COM single-threaded apartment thread blocks, the thread can wake up internally to pump messages. For example, the blocked thread will wake to process a Windows message sent from another thread. This is done to support COM interoperability. For most applications, this is not a problem—in fact, it is a good thing. However, if your code takes another thread synchronization lock during the processing of the message, then deadlock could occur. As you'll see in Chapter 30, all the hybrid locks call these methods internally, so the same potential benefit or problem exists when using the hybrid locks as well.

The versions of the `WaitOne` and `WaitAll` that do not accept a timeout parameter should be prototyped as having a `void` return type, not `Boolean`. The reason is because these methods will return only `true` because the implied timeout is infinite (`System.Threading.Timeout.Infinite`). When you call any of these methods, you do not need to check their return value.

As already mentioned, the `AutoResetEvent`, `ManualResetEvent`, `Semaphore`, and `Mutex` classes are all derived from `WaitHandle`, so they inherit `WaitHandle`'s methods and their behavior. However, these classes introduce additional methods of their own, and I'll address those now.

First, the constructors for all of these classes internally call the Win32 `CreateEvent` (passing `FALSE` for the `bManualReset` parameter) or `CreateEvent` (passing `TRUE` for the `bManualReset` parameter), `CreateSemaphore`, or `CreateMutex` functions. The handle value returned from all of these calls is saved in a private `SafeWaitHandle` field defined inside the `WaitHandle` base class.

Second, the `EventWaitHandle`, `Semaphore`, and `Mutex` classes all offer static `OpenExisting` methods, which internally call the Win32 `OpenEvent`, `OpenSemaphore`, or `OpenMutex` functions, passing a `String` argument that identifies an existing named kernel object. The handle value returned from all of these functions is saved in a newly constructed object that is returned from the `OpenExisting` method. If no kernel object exists with the specified name, a `WaitHandleCannotBeOpenedException` is thrown.

A common usage of the kernel-mode constructs is to create the kind of application that allows only one instance of itself to execute at any given time. Examples of single-instance applications are Microsoft Outlook, Windows Live Messenger, Windows Media Player, and Windows Media Center. Here is how to implement a single-instance application.

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
      Boolean createdNew;

      // Try to create a kernel object with the specified name
      using (new Semaphore(0, 1, "SomeUniqueStringIdentifyingMyApp", out createdNew)) {
         if (createdNew) {
            // This thread created the kernel object so no other instance of this
            // application must be running. Run the rest of the application here...
         } else {
            // This thread opened an existing kernel object with the same string name;
            // another instance of this application must be running now.
            // There is nothing to do in here, let's just return from Main to terminate
            // this second instance of the application.
         }
      }
   }
}
```

In this code, I am using a `Semaphore`, but it would work just as well if I had used an `EventWaitHandle` or a `Mutex` because I'm not actually using the thread synchronization behavior that the object offers. However, I am taking advantage of some thread synchronization behavior that the kernel offers when creating any kind of kernel object. Let me explain how the preceding code works. Let's say that two instances of this process are started at exactly the same time. Each process will have its own thread, and both threads will attempt to create a `Semaphore` with the same string name ("`SomeUniqueStringIdentifyingMyApp`," in my example). The Windows kernel ensures that only one thread actually creates a kernel object with the specified name; the thread that created the object will have its `createdNew` variable set to `true`.

For the second thread, Windows will see that a kernel object with the specified name already exists; the second thread does not get to create another kernel object with the same name, although if this thread continues to run, it can access the same kernel object as the first process's thread. This is how threads in different processes can communicate with each other via a single kernel object. However, in this example, the second process's thread sees that its `createdNew` variable is set to `false`. This thread now knows that another instance of this process is running, and the second instance of the process exits immediately.

## Event Constructs

Events are simply `Boolean` variables maintained by the kernel. A thread waiting on an event blocks when the event is `false` and unblocks when the event is `true`. There are two kinds of events. When an auto-reset event is `true`, it wakes up just one blocked thread, because the kernel *automatically resets* the event back to `false` after unblocking the first thread. When a manual-reset event is `true`, it unblocks all threads waiting for it because the kernel does not automatically reset the event back to `false`; your code must *manually reset* the event back to `false`. The classes related to events look like this.

```
public class EventWaitHandle : WaitHandle {
   public Boolean Set();    // Sets Boolean to true; always returns true
   public Boolean Reset();  // Sets Boolean to false; always returns true
}

public sealed class AutoResetEvent : EventWaitHandle {
   public AutoResetEvent(Boolean initialState);
}

public sealed class ManualResetEvent : EventWaitHandle {
   public ManualResetEvent(Boolean initialState);
}
```

Using an auto-reset event, we can easily create a thread synchronization lock whose behavior is similar to the `SimpleSpinLock` class I showed earlier.

```
internal sealed class SimpleWaitLock : IDisposable {
   private readonly AutoResetEvent m_available;

   public SimpleWaitLock() {
      m_available = new AutoResetEvent(true); // Initially free
   }

   public void Enter() {
      // Block in kernel until resource available
      m_available.WaitOne();
   }

   public void Leave() {
      // Let another thread access the resource
      m_available.Set();
   }

   public void Dispose() { m_available.Dispose(); }
}
```

You would use this `SimpleWaitLock` exactly the same way that you'd use the `SimpleSpinLock`. In fact, the external behavior is exactly the same; however, the performance of the two locks is radically different. When there is no contention on the lock, the `SimpleWaitLock` is much slower than the `SimpleSpinLock`, because every call to `SimpleWaitLock`'s `Enter` and `Leave` methods forces the calling thread to transition from managed code to the kernel and back—which is bad. But when there is contention, the losing thread is blocked by the kernel and is not spinning and wasting CPU

cycles—which is good. Note also that constructing the `AutoResetEvent` object and calling `Dispose` on it also causes managed to kernel transitions, affecting performance negatively. These calls usually happen rarely, so they are not something to be too concerned about.

To give you a better feel for the performance differences, I wrote the following code.

```
public static void Main() {
   Int32 x = 0;
   const Int32 iterations = 10000000;  // 10 million

   // How long does it take to increment x 10 million times?
   Stopwatch sw = Stopwatch.StartNew();
   for (Int32 i = 0; i < iterations; i++) {
      x++;
   }
   Console.WriteLine("Incrementing x: {0:N0}", sw.ElapsedMilliseconds);

   // How long does it take to increment x 10 million times
   // adding the overhead of calling a method that does nothing?
   sw.Restart();
   for (Int32 i = 0; i < iterations; i++) {
      M(); x++; M();
   }
   Console.WriteLine("Incrementing x in M: {0:N0}", sw.ElapsedMilliseconds);

   // How long does it take to increment x 10 million times
   // adding the overhead of calling an uncontended SimpleSpinLock?
   SpinLock sl = new SpinLock(false);
   sw.Restart();
   for (Int32 i = 0; i < iterations; i++) {
      Boolean taken = false; sl.Enter(ref taken); x++; sl.Exit();
   }
   Console.WriteLine("Incrementing x in SpinLock: {0:N0}", sw.ElapsedMilliseconds);

   // How long does it take to increment x 10 million times
   // adding the overhead of calling an uncontended SimpleWaitLock?
   using (SimpleWaitLock swl = new SimpleWaitLock()) {
      sw.Restart();
      for (Int32 i = 0; i < iterations; i++) {
         swl.Enter(); x++; swl.Leave();
      }
      Console.WriteLine("Incrementing x in SimpleWaitLock: {0:N0}", sw.ElapsedMilliseconds);
   }
}

[MethodImpl(MethodImplOptions.NoInlining)]
private static void M() { /* This method does nothing but return */ }
```

When I run the preceding code, I get the following output.

```
Incrementing x: 8                          Fastest
Incrementing x in M: 69                    ~9x slower
Incrementing x in SpinLock: 164            ~21x slower
Incrementing x in SimpleWaitLock: 8,854    ~1,107x slower
```

As you can clearly see, just incrementing x took only 8 milliseconds. To call empty methods before and after incrementing x made the operation take nine times longer! Then, executing code in a method that uses a user-mode construct caused the code to run 21 (164 / 8) times slower. But now, see how much slower the program ran using a kernel-mode construct: 1,107 (8,854 / 8) times slower! So, if you can avoid thread synchronization, you should. If you need thread synchronization, then try to use the user-mode constructs. Always try to avoid the kernel-mode constructs.

## Semaphore Constructs

Semaphores are simply `Int32` variables maintained by the kernel. A thread waiting on a semaphore blocks when the semaphore is 0 and unblocks when the semaphore is greater than 0. When a thread waiting on a semaphore unblocks, the kernel automatically subtracts 1 from the semaphore's count. Semaphores also have a maximum `Int32` value associated with them, and the current count is never allowed to go over the maximum count. Here is what the `Semaphore` class looks like.

```
public sealed class Semaphore : WaitHandle {
   public Semaphore(Int32 initialCount, Int32 maximumCount);
   public Int32 Release();   // Calls Release(1); returns previous count
   public Int32 Release(Int32 releaseCount);   // Returns previous count
}
```

So now let me summarize how these three kernel-mode primitives behave:

- When multiple threads are waiting on an auto-reset event, setting the event causes *only one* thread to become unblocked.
- When multiple threads are waiting on a manual-reset event, setting the event causes *all* threads to become unblocked.
- When multiple threads are waiting on a semaphore, releasing the semaphore causes `releaseCount` threads to become unblocked (where `releaseCount` is the argument passed to `Semaphore`'s `Release` method).

Therefore, an auto-reset event behaves very similarly to a semaphore whose maximum count is 1. The difference between the two is that `Set` can be called multiple times consecutively on an auto-reset event, and still only one thread will be unblocked, whereas calling `Release` multiple times consecutively on a semaphore keeps incrementing its internal count, which could unblock many threads. By the way, if you call `Release` on a semaphore too many times, causing its count to exceed its maximum count, then `Release` will throw a `SemaphoreFullException`.

Using a semaphore, we can re-implement the `SimpleWaitLock` as follows, so that it gives multiple threads concurrent access to a resource (which is not necessarily a safe thing to do unless all threads access the resource in a read-only fashion).

```
public sealed class SimpleWaitLock : IDisposable {
   private readonly Semaphore m_available;

   public SimpleWaitLock(Int32 maxConcurrent) {
      m_available = new Semaphore(maxConcurrent, maxConcurrent);
   }

   public void Enter() {
      // Block in kernel until resource available
      m_available.WaitOne();
   }

   public void Leave() {
      // Let another thread access the resource
      m_available.Release(1);
   }

   public void Dispose() { m_available.Close(); }
}
```

## Mutex Constructs

A `Mutex` represents a mutual-exclusive lock. It works similar to an `AutoResetEvent` or a `Semaphore` with a count of `1` because all three constructs release only one waiting thread at a time. The following shows what the `Mutex` class looks like.

```
public sealed class Mutex : WaitHandle {
   public Mutex();
   public void ReleaseMutex();
}
```

Mutexes have some additional logic in them, which makes them more complex than the other constructs. First, `Mutex` objects record which thread obtained it by querying the calling thread's `Int32` ID. When a thread calls `ReleaseMutex`, the `Mutex` makes sure that the calling thread is the same thread that obtained the `Mutex`. If the calling thread is not the thread that obtained the `Mutex`, then the `Mutex` object's state is unaltered and `ReleaseMutex` throws a `System.ApplicationException`. Also, if a thread owning a `Mutex` terminates for any reason, then some thread waiting on the `Mutex` will be awakened by having a `System.Threading.AbandonedMutexException` thrown. Usually, this exception will go unhandled, terminating the whole process. This is good because a thread acquired the `Mutex` and it is likely that the thread terminated before it finished updating the data that the `Mutex` was protecting. If a thread catches `AbandonedMutexException`, then it could attempt to access the corrupt data, leading to unpredictable results and security problems.

Second, `Mutex` objects maintain a recursion count indicating how many times the owning thread owns the `Mutex`. If a thread currently owns a `Mutex` and then that thread waits on the `Mutex` again, the recursion count is incremented and the thread is allowed to continue running. When that thread calls `ReleaseMutex`, the recursion count is decremented. Only when the recursion count becomes 0 can another thread become the owner of the `Mutex`.

Most people do not like this additional logic. The problem is that these "features" have a cost associated with them. The `Mutex` object needs more memory to hold the additional thread ID and recursion count information. And, more importantly, the `Mutex` code has to maintain this information, which makes the lock slower. If an application needs or wants these additional features, then the application code could have done this itself; the code doesn't have to be built into the `Mutex` object. For this reason, a lot of people avoid using `Mutex` objects.

Usually a recursive lock is needed when a method takes a lock and then calls another method that also requires the lock, as the following code demonstrates.

```
internal class SomeClass : IDisposable {
   private readonly Mutex m_lock = new Mutex();

   public void Method1() {
      m_lock.WaitOne();
      // Do whatever...
      Method2();  // Method2 recursively acquires the lock
      m_lock.ReleaseMutex();
   }

   public void Method2() {
      m_lock.WaitOne();
      // Do whatever...
      m_lock.ReleaseMutex();
   }

   public void Dispose() { m_lock.Dispose(); }
}
```

In the preceding code, code that uses a `SomeClass` object could call `Method1`, which acquires the `Mutex`, performs some thread-safe operation, and then calls `Method2`, which also performs some thread-safe operation. Because `Mutex` objects support recursion, the thread will acquire the lock twice and then release it twice before another thread can own the `Mutex`. If `SomeClass` has used an `AutoResetEvent` instead of a `Mutex`, then the thread would block when it called `Method2`'s `WaitOne` method.

If you need a recursive lock, then you could create one easily by using an `AutoResetEvent`.

```
internal sealed class RecursiveAutoResetEvent : IDisposable {
   private AutoResetEvent m_lock = new AutoResetEvent(true);
   private Int32 m_owningThreadId = 0;
   private Int32 m_recursionCount = 0;

   public void Enter() {
      // Obtain the calling thread's unique Int32 ID
      Int32 currentThreadId = Thread.CurrentThread.ManagedThreadId;

      // If the calling thread owns the lock, increment the recursion count
      if (m_owningThreadId == currentThreadId) {
         m_recursionCount++;
         return;
      }
```

```
      // The calling thread doesn't own the lock, wait for it
      m_lock.WaitOne();

      // The calling now owns the lock, initialize the owning thread ID & recursion count
      m_owningThreadId = currentThreadId;
      m_recursionCount = 1;
   }

   public void Leave() {
      // If the calling thread doesn't own the lock, we have an error
      if (m_owningThreadId != Thread.CurrentThread.ManagedThreadId)
         throw new InvalidOperationException();

      // Subtract 1 from the recursion count
      if (--m_recursionCount == 0) {
         // If the recursion count is 0, then no thread owns the lock
         m_owningThreadId = 0;
         m_lock.Set();   // Wake up 1 waiting thread (if any)
      }
   }

   public void Dispose() { m_lock.Dispose(); }
}
```

Although the behavior of the `RecursiveAutoResetEvent` class is identical to that of the `Mutex` class, a `RecursiveAutoResetEvent` object will have far superior performance when a thread tries to acquire the lock recursively, because all the code that is required to track thread ownership and recursion is now in managed code. A thread has to transition into the Windows kernel only when first acquiring the `AutoResetEvent` or when finally relinquishing it to another thread.