CHAPTER 30

Hybrid Thread Synchronization Constructs

In this chapter:

In Chapter 29, "Primitive Thread Synchronization Constructs," I discussed the primitive user-mode and kernel-mode thread synchronization constructs. From these primitive constructs, all other thread synchronization constructs can be built. Typically, other thread synchronization constructs are built by combining the user-mode and kernel-mode constructs, and I call these *hybrid thread synchronization constructs*. Hybrid constructs provide the performance benefit of the primitive user-mode constructs when there is no thread contention. Hybrid constructs also use the primitive kernel-mode constructs to provide the benefit of not spinning (wasting CPU time) when multiple threads are contending for the construct at the same time. Because, in most applications, threads are rarely contending for a construct at the same time, the performance improvements can help your application greatly.

In this chapter, I will first show how hybrid constructs are built from the various primitive constructs. Then, I will introduce you to many of the hybrid constructs that ship with the Framework Class Library (FCL), describe their behavior, and give some insight as to how to use these constructs correctly. I will also mention some constructs that I have created and make available for free in Wintellect's Power Threading library, which can be downloaded from *http://Wintellect.com/PowerThreading.aspx*.

Toward the end of the chapter, I show how to minimize resource usage and improve performance by using the FCL's concurrent collection classes instead of using some of the hybrid constructs. And finally, I discuss asynchronous synchronization constructs, which allow you to synchronize access to a resource without blocking any threads, thereby reducing resource consumption while improving scalability.

A Simple Hybrid Lock

So, without further ado, let me start off by showing you an example of a hybrid thread synchronization lock.

```
internal sealed class SimpleHybridLock : IDisposable {
   // The Int32 is used by the primitive user-mode constructs (Interlocked methods)
   private Int32 m_waiters = 0;

   // The AutoResetEvent is the primitive kernel-mode construct
   private readonly AutoResetEvent m_waiterLock = new AutoResetEvent(false);

   public void Enter() {
      // Indicate that this thread wants the lock
      if (Interlocked.Increment(ref m_waiters) == 1)
         return; // Lock was free, no contention, just return

      // Another thread has the lock (contention), make this thread wait
      m_waiterLock.WaitOne();  // Bad performance hit here
      // When WaitOne returns, this thread now has the lock
   }

   public void Leave() {
      // This thread is releasing the lock
      if (Interlocked.Decrement(ref m_waiters) == 0)
         return; // No other threads are waiting, just return

      // Other threads are waiting, wake 1 of them
      m_waiterLock.Set();  // Bad performance hit here
   }

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

The `SimpleHybridLock` contains two fields: an `Int32`, which will be manipulated via the primitive user-mode constructs, and an `AutoResetEvent`, which is a primitive kernel-mode construct. To get great performance, the lock tries to use the `Int32` and avoid using the `AutoResetEvent` as much as possible. Just constructing a `SimpleHybridLock` object causes the `AutoResetEvent` to be created, and this is a massive performance hit compared to the overhead associated with the `Int32` field. Later in this chapter, we'll see another hybrid construct (`AutoResetEventSlim`) that avoids the performance hit of creating the `AutoResetEvent` until the first time contention is detected from multiple threads accessing the lock at the same time. The `Dispose` method closes the `AutoResetEvent`, and this is also a big performance hit.

Although it would be nice to improve the performance of constructing and disposing of a `SimpleHybridLock` object, it would be better to focus on the performance of its `Enter` and `Leave` methods because these methods tend to be called many, many times over the object's lifetime. Let's focus on these methods now.

The first thread to call Enter causes Interlocked.Increment to add one to the m_waiters field, making its value 1. This thread sees that there were zero threads waiting for this lock, so the thread gets to return from its call to Enter. The thing to appreciate here is that the thread acquired the lock very quickly. Now, if another thread comes along and calls Enter, this second thread increments m_waiters to 2 and sees that another thread has the lock, so this thread blocks by calling WaitOne using the AutoResetEvent. Calling WaitOne causes the thread to transition into the Windows' kernel, and this is a big performance hit. However, the thread must stop running anyway, so it is not too bad to have a thread waste some time to stop completely. The good news is that the thread is now blocked, and so it is not wasting CPU time by spinning on the CPU, which is what the SimpleSpinLock's Enter method, introduced in Chapter 29, does.

Now let's look at the Leave method. When a thread calls Leave, Interlocked.Decrement is called to subtract 1 from the m_waiters field. If m_waiters is now 0, then no other threads are blocked inside a call to Enter and the thread calling Leave can simply return. Again, think about how fast this is: leaving a lock means that a thread subtracts 1 from an Int32, performs a quick if test, and then returns! On the other hand, if the thread calling Leave sees that m_waiters was not 1, then the thread knows that there is contention and that there is at least one other thread blocked in the kernel. This thread must wake up one (and only one) of the blocked threads. It does this by calling Set on AutoResetEvent. This is a performance hit, because the thread must transition into the kernel and back, but this transition occurs only when there was contention. Of course, AutoResetEvent ensures that only one blocked thread wakes up; any other threads blocked on the AutoResetEvent will continue to block until the newly unblocked thread eventually calls Leave.

Note In reality, any thread could call Leave at any time because the Enter method does not keep a record of which thread successfully acquired the lock. Adding the field and code to maintain this information is easy to do, but it would increase the memory required for the lock object itself and hurt performance of the Enter and Leave methods because they would have to manipulate this field. I would rather have a fast-performing lock and make sure that my code uses it the right way. You'll notice that events and semaphores do not maintain this kind of information; only mutexes do.

Spinning, Thread Ownership, and Recursion

Because transitions into the kernel incur such a big performance hit and threads tend to hold on to a lock for very short periods of time, an application's overall performance can be improved by having a thread spin in user mode for a little while before having the thread transition to kernel mode. If the lock that the thread is waiting for becomes available while spinning, then the transition to kernel mode is avoided.

In addition, some locks impose a limitation where the thread that acquires the lock must be the thread that releases the lock. And some locks allow the currently owning thread to own the lock

recursively. The Mutex lock is an example of a lock that has these characteristics.[1] Using some fancy logic, it is possible to build a hybrid lock that offers spinning, thread ownership, and recursion. Here is what the code looks like.

```
internal sealed class AnotherHybridLock : IDisposable {
   // The Int32 is used by the primitive user-mode constructs (Interlocked methods)
   private Int32 m_waiters = 0;

   // The AutoResetEvent is the primitive kernel-mode construct
   private AutoResetEvent m_waiterLock = new AutoResetEvent(false);

   // This field controls spinning in an effort to improve performance
   private Int32 m_spincount = 4000;   // Arbitrarily chosen count

   // These fields indicate which thread owns the lock and how many times it owns it
   private Int32 m_owningThreadId = 0, m_recursion = 0;

   public void Enter() {
      // If calling thread already owns the lock, increment recursion count and return
      Int32 threadId = Thread.CurrentThread.ManagedThreadId;
      if (threadId == m_owningThreadId) { m_recursion++; return; }

      // The calling thread doesn't own the lock, try to get it
      SpinWait spinwait = new SpinWait();
      for (Int32 spinCount = 0; spinCount < m_spincount; spinCount++) {
         // If the lock was free, this thread got it; set some state and return
         if (Interlocked.CompareExchange(ref m_waiters, 1, 0) == 0) goto GotLock;

         // Black magic: give other threads a chance to run
         // in hopes that the lock will be released
         spinwait.SpinOnce();
      }

      // Spinning is over and the lock was still not obtained, try one more time
      if (Interlocked.Increment(ref m_waiters) > 1) {
         // Still contention, this thread must wait
         m_waiterLock.WaitOne(); // Wait for the lock; performance hit
         // When this thread wakes, it owns the lock; set some state and return
      }

   GotLock:
      // When a thread gets the lock, we record its ID and
      // indicate that the thread owns the lock once
      m_owningThreadId = threadId; m_recursion = 1;
   }

   public void Leave() {
      // If the calling thread doesn't own the lock, there is a bug
      Int32 threadId = Thread.CurrentThread.ManagedThreadId;
      if (threadId != m_owningThreadId)
         throw new SynchronizationLockException("Lock not owned by calling thread");
```

[1] Threads do not spin when waiting on a Mutex object because the Mutex's code is in the kernel. This means that the thread had to have already transitioned into the kernel to check the Mutex's state.

```
      // Decrement the recursion count. If this thread still owns the lock, just return
      if (--m_recursion > 0) return;

      m_owningThreadId = 0;   // No thread owns the lock now

      // If no other threads are waiting, just return
      if (Interlocked.Decrement(ref m_waiters) == 0)
         return;

      // Other threads are waiting, wake 1 of them
      m_waiterLock.Set();     // Bad performance hit here
   }

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

As you can see, adding extra behavior to the lock increases the number of fields it has, which increases its memory consumption. The code is also more complex, and this code must execute, which decreases the lock's performance. In Chapter 29's "Event Constructs" section, I compared the performance of incrementing an `Int32` without any locking, with a primitive user-mode construct, and with a kernel-mode construct. I repeat the results of those performance tests here and I include the results of using the `SimpleHybridlock` and the `AnotherHybridLock`. The results are in fastest to slowest order.

```
Incrementing x: 8                           Fastest
Incrementing x in M: 69                     ~9x slower
Incrementing x in SpinLock: 164             ~21x slower
Incrementing x in SimpleHybridLock: 164     ~21x slower (similar to SpinLock)
Incrementing x in AnotherHybridLock: 230    ~29x slower (due to ownership/recursion)
Incrementing x in SimpleWaitLock: 8,854     ~1,107x slower
```

It is worth noting that the `AnotherHybridLock` hurts performance as compared to using the `SimpleHybridLock`. This is due to the additional logic and error checking required managing the thread ownership and recursion behaviors. As you see, every behavior added to a lock impacts its performance.

Hybrid Constructs in the Framework Class Library

The FCL ships with many hybrid constructs that use fancy logic to keep your threads in user mode, improving your application's performance. Some of these hybrid constructs also avoid creating the kernel-mode construct until the first time threads contend on the construct. If threads never contend on the construct, then your application avoids the performance hit of creating the object and also avoids allocating memory for the object. A number of the constructs also support the use of a `CancellationToken` (discussed in Chapter 27, "Compute-Bound Asynchronous Operations") so that a thread can forcibly unblock other threads that might be waiting on the construct. In this section, I introduce you to these hybrid constructs.

The `ManualResetEventSlim` and `SemaphoreSlim` Classes

The first two hybrid constructs are `System.Threading.ManualResetEventSlim` and `System.Threading.SemaphoreSlim`.[2] These constructs work exactly like their kernel-mode counterparts, except that both employ spinning in user mode, and they both defer creating the kernel-mode construct until the first time contention occurs. Their `Wait` methods allow you to pass a timeout and a `CancellationToken`. Here is what these classes look like (some method overloads are not shown).

```
public class ManualResetEventSlim : IDisposable {
   public ManualResetEventSlim(Boolean initialState, Int32 spinCount);
   public void Dispose();
   public void Reset();
   public void Set();
   public Boolean Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken);

   public Boolean IsSet { get; }
   public Int32 SpinCount { get; }
   public WaitHandle WaitHandle { get; }
}
public class SemaphoreSlim : IDisposable {
   public SemaphoreSlim(Int32 initialCount, Int32 maxCount);
   public void Dispose();
   public Int32 Release(Int32 releaseCount);
   public Boolean Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken);

   // Special method for use with async and await (see Chapter 28)
   public Task<Boolean> WaitAsync(Int32 millisecondsTimeout, CancellationToken
cancellationToken);

   public Int32 CurrentCount { get; }
   public WaitHandle AvailableWaitHandle { get; }
}
```

The `Monitor` Class and Sync Blocks

Probably the most-used hybrid thread synchronization construct is the `Monitor` class, which provides a mutual-exclusive lock supporting spinning, thread ownership, and recursion. This is the most-used construct because it has been around the longest, C# has a built-in keyword to support it, the just-in-time (JIT) compiler has built-in knowledge of it, and the common language runtime (CLR) itself uses it on your application's behalf. However, as you'll see, there are many problems with this construct, making it easy to produce buggy code. I'll start by explaining the construct, and then I'll show the problems and some ways to work around these problems.

Every object on the heap can have a data structure, called a *sync block*, associated with it. A sync block contains fields similar to that of the `AnotherHybridLock` class that appeared earlier in this chapter. Specifically, it has fields for a kernel object, the owning thread's ID, a recursion count, and a waiting threads count. The `Monitor` class is a `static` class whose methods accept a reference to

[2] Although there is no `AutoResetEventSlim` class, in many situations you can construct a `SemaphoreSlim` object with a `maxCount` of 1.

any heap object, and these methods manipulate the fields in the specified object's sync block. Here is what the most commonly used methods of the `Monitor` class look like.

```
public static class Monitor {
   public static void Enter(Object obj);
   public static void Exit(Object obj);

   // You can also specify a timeout when entered the lock (not commonly used):
   public static Boolean TryEnter(Object obj, Int32 millisecondsTimeout);

   // I'll discuss the lockTaken argument later
   public static void Enter(Object obj, ref Boolean lockTaken);
   public static void TryEnter(Object obj, Int32 millisecondsTimeout, ref Boolean lockTaken);
}
```

Now obviously, associating a sync block data structure with every object in the heap is quite wasteful, especially because most objects' sync blocks are never used. To reduce memory usage, the CLR team uses a more efficient way to offer the functionality just described. Here's how it works: when the CLR initializes, it allocates an array of sync blocks in native heap. As discussed elsewhere in this book, whenever an object is created in the heap, it gets two additional overhead fields associated with it. The first overhead field, the type object pointer, contains the memory address of the type's type object. The second overhead field, the sync block index, contains an integer index into the array of sync blocks.

When an object is constructed, the object's sync block index is initialized to -1, which indicates that it doesn't refer to any sync block. Then, when `Monitor.Enter` is called, the CLR finds a free sync block in the array and sets the object's sync block index to refer to the sync block that was found. In other words, sync blocks are associated with an object on the fly. When `Exit` is called, it checks to see whether there are any more threads waiting to use the object's sync block. If there are no threads waiting for it, the sync block is free, `Exit` sets the object's sync block index back to -1, and the free sync block can be associated with another object in the future.

Figure 30-1 shows the relationship between objects in the heap, their sync block indexes, and elements in the CLR's sync block array. Object-A, Object-B, and Object-C all have their type object pointer member set to refer to Type-T (a type object). This means that all three objects are of the same type. As discussed in Chapter 4, "Type Fundamentals," a type object is also an object in the heap, and like all other objects, a type object has the two overhead members: a sync block index and a type object pointer. This means that a sync block can be associated with a type object and a reference to a type object can be passed to `Monitor`'s methods. By the way, the sync block array is able to create more sync blocks if necessary, so you shouldn't worry about the system running out of sync blocks if many objects are being synchronized simultaneously.

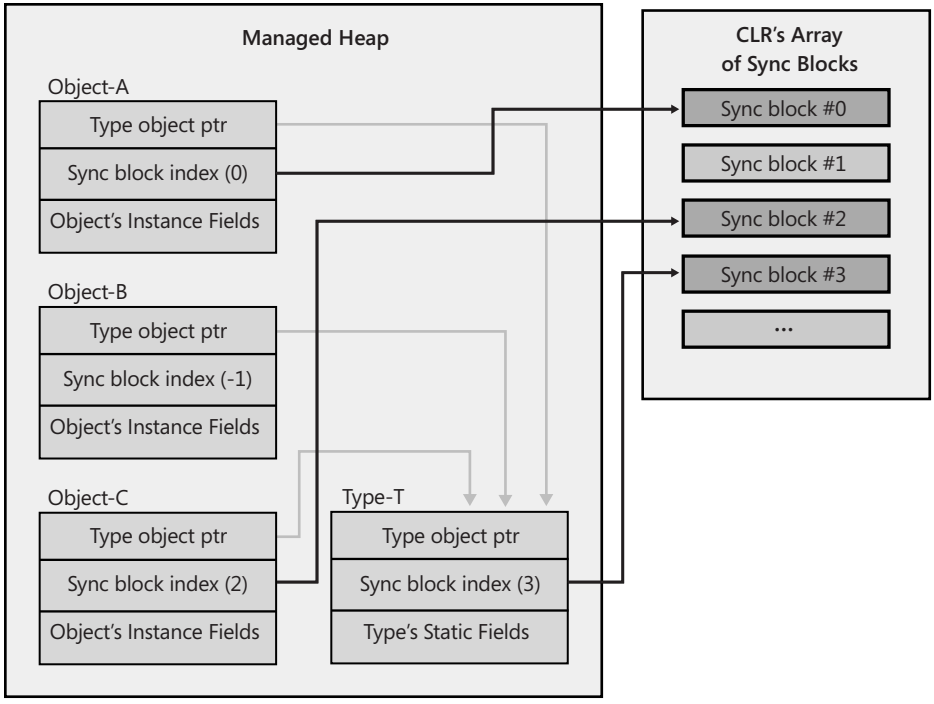

FIGURE 30-1 Objects in the heap (including type objects) can have their sync block indexes refer to an entry in the CLR's sync block array.

Here is some code that demonstrates how the `Monitor` class was originally intended to be used.

```
internal sealed class Transaction {
   private DateTime m_timeOfLastTrans;

   public void PerformTransaction() {
      Monitor.Enter(this);
      // This code has exclusive access to the data...
      m_timeOfLastTrans = DateTime.Now;
      Monitor.Exit(this);
   }

   public DateTime LastTransaction {
      get {
         Monitor.Enter(this);
         // This code has exclusive access to the data...
         DateTime temp = m_timeOfLastTrans;
         Monitor.Exit(this);
         return temp;
      }
   }
}
```

On the surface, this seems simple enough, but there is something wrong with this code. The problem is that each object's sync block index is implicitly public. The following code demonstrates the impact of this.

```
public static void SomeMethod() {
   var t = new Transaction();
   Monitor.Enter(t); // This thread takes the object's public lock

   // Have a thread pool thread display the LastTransaction time
   // NOTE: The thread pool thread blocks until SomeMethod calls Monitor.Exit!
   ThreadPool.QueueUserWorkItem(o => Console.WriteLine(t.LastTransaction));

   // Execute some other code here...
   Monitor.Exit(t);
}
```

In this code, the thread executing `SomeMethod` calls `Monitor.Enter`, taking the `Transaction` object's publicly exposed lock. When the thread pool thread queries the `LastTransaction` property, this property also calls `Monitor.Enter` to acquire the same lock, causing the thread pool thread to block until the thread executing `SomeMethod` calls `Monitor.Exit`. Using a debugger, you can determine that the thread pool thread is blocked inside the `LastTransaction` property, but it is very hard to determine which other thread has the lock. If you do somehow figure out which thread has the lock, then you have to figure out what code caused it to take the lock. This is very difficult, and even worse, if you do figure it out, then the code might not be code that you have control over and you might not be able to modify this code to fix the problem. Therefore, my suggestion to you is to always use a private lock instead. Here's how I'd fix the `Transaction` class.

```
internal sealed class Transaction {
   private readonly Object m_lock = new Object(); // Each transaction has a PRIVATE lock now
   private DateTime m_timeOfLastTrans;

   public void PerformTransaction() {
      Monitor.Enter(m_lock);     // Enter the private lock
      // This code has exclusive access to the data...
      m_timeOfLastTrans = DateTime.Now;
      Monitor.Exit(m_lock);      // Exit the private lock
   }

   public DateTime LastTransaction {
      get {
         Monitor.Enter(m_lock);  // Enter the private lock
         // This code has exclusive access to the data...
         DateTime temp = m_timeOfLastTrans;
         Monitor.Exit(m_lock);   // Exit the private lock
         return temp;
      }
   }
}
```

If `Transaction`'s members were `static`, then simply make the `m_lock` field `static`, too, and now the `static` members are thread safe.

It should be clear from this discussion that `Monitor` should not have been implemented as a `static` class; it should have been implemented like all the other locks: a class you instantiate and call instance methods on. In fact, `Monitor` has many other problems associated with it that are all because it is a `static` class. Here is a list of additional problems:

- A variable can refer to a proxy object if the type of object it refers to is derived from the `System.MarshalByRefObject` class (discussed in Chapter 22, "CLR Hosting and AppDomains"). When you call `Monitor`'s methods, passing a reference to a proxy object, you are locking the proxy object, not the actual object that the proxy refers to.
- If a thread calls `Monitor.Enter`, passing it a reference to a type object that has been loaded domain neutral (discussed in Chapter 22), the thread is taking a lock on that type across all AppDomains in the process. This is a known bug in the CLR that violates the isolation that AppDomains are supposed to provide. The bug is difficult to fix in a high-performance way, so it never gets fixed. The recommendation is to never pass a reference to a type object into `Monitor`'s methods.
- Because strings can be interned (as discussed in Chapter 14, "Chars, Strings, and Working with Text"), two completely separate pieces of code could unknowingly get references to a single `String` object in memory. If they pass the reference to the `String` object into `Monitor`'s methods, then the two separate pieces of code are now synchronizing their execution with each other unknowingly.
- When passing a string across an AppDomain boundary, the CLR does not make a copy of the string; instead, it simply passes a reference to the string into the other AppDomain. This improves performance, and in theory, it should be OK because `String` objects are immutable. However, like all objects, `String` objects have a sync block index associated with them, which is mutable, and this allows threads in different AppDomains to synchronize with each other unknowingly. This is another bug in CLR's AppDomain isolation story. The recommendation is never to pass `String` references to `Monitor`'s methods.
- Because `Monitor`'s methods take an `Object`, passing a value type causes the value type to get boxed, resulting in the thread taking a lock on the boxed object. Each time `Monitor.Enter` is called, a lock is taken on a completely different object and you get no thread synchronization at all.
- Applying the `[MethodImpl(MethodImplOptions.Synchronized)]` attribute to a method causes the JIT compiler to surround the method's native code with calls to `Monitor.Enter` and `Monitor.Exit`. If the method is an instance method, then `this` is passed to these methods, locking the implicitly public lock. If the method is static, then a reference to the type's type object is passed to these methods, potentially locking a domain-neutral type. The recommendation is to never use this attribute.

- When calling a type's type constructor (discussed in Chapter 8, "Methods"), the CLR takes a lock on the type's type object to ensure that only one thread initializes the type object and its static fields. Again, this type could be loaded domain neutral, causing a problem. For example, if the type constructor's code enters an infinite loop, then the type is unusable by all AppDomains in the process. The recommendation here is to avoid type constructors as much as possible or least keep them short and simple.

Unfortunately, the story gets worse. Because it is so common for developers to take a lock, do some work, and then release the lock within a single method, the C# language offers simplified syntax via its `lock` keyword. Suppose that you write a method like this.

```
private void SomeMethod() {
   lock (this) {
      // This code has exclusive access to the data...
   }
}
```

It is equivalent to having written the method like this.

```
private void SomeMethod() {
   Boolean lockTaken = false;
   try {
      // An exception (such as ThreadAbortException) could occur here...
      Monitor.Enter(this, ref lockTaken);
      // This code has exclusive access to the data...
   }
   finally {
      if (lockTaken) Monitor.Exit(this);
   }
}
```

The first problem here is that the C# team felt that they were doing you a favor by calling `Monitor.Exit` in a `finally` block. Their thinking was that this ensures that the lock is always released no matter what happens inside the `try` block. However, this is not a good thing. If an exception occurs inside the `try` block while changing state, then the state is now corrupted. When the lock is exited in the `finally` block, another thread will now start manipulating the corrupted state. It is better to have your application hang than it is to continue running with a corrupted state and potential security holes. The second problem is that entering and leaving a `try` block decreases the performance of the method. And some JIT compilers won't inline a method that contains a `try` block in it, which decreases performance even more. So now we have slower code that lets threads access corrupted state.[3] The recommendation is not to use C#'s `lock` statement.

Now we get to the `Boolean lockTaken` variable. Here is the problem that this variable is trying to solve. Let's say that a thread enters the `try` block and before calling `Monitor.Enter`, the thread is aborted (as discussed in Chapter 22). Now the `finally` block is called, but its code should not exit the lock. The `lockTaken` variable solves this problem. It is initialized to `false`, which assumes that the lock has not been entered into. Then, if `Monitor.Enter` is called and successfully takes the

[3] By the way, while still a performance hit, it is safe to release a `lock` in a `finally` block if the code in the `try` block reads the state without attempting to modify it.

lock, it sets `lockTaken` to `true`. The `finally` block examines `lockTaken` to know whether to call `Monitor.Exit` or not. By the way, the `SpinLock` structure also supports this `lockTaken` pattern.

The `ReaderWriterLockSlim` Class

It is common to have threads simply read the contents of some data. If this data is protected by a mutual exclusive lock (like the `SimpleSpinLock`, `SimpleWaitLock`, `SimpleHybridLock`, `AnotherHybridLock`, `SpinLock`, `Mutex`, or `Monitor`), then if multiple threads attempt this access concurrently, only one thread gets to run and all the other threads are blocked, which can reduce scalability and throughput in your application substantially. However, if all the threads want to access the data in a read-only fashion, then there is no need to block them at all; they should all be able to access the data concurrently. On the other hand, if a thread wants to modify the data, then this thread needs exclusive access to the data. The `ReaderWriterLockSlim` construct encapsulates the logic to solve this problem. Specifically, the construct controls threads like this:

- When one thread is writing to the data, all other threads requesting access are blocked.
- When one thread is reading from the data, other threads requesting read access are allowed to continue executing, but threads requesting write access are blocked.
- When a thread writing to the data has completed, either a single writer thread is unblocked so it can access the data or all the reader threads are unblocked so that all of them can access the data concurrently. If no threads are blocked, then the lock is free and available for the next reader or writer thread that wants it.
- When all threads reading from the data have completed, a single writer thread is unblocked so it can access the data. If no threads are blocked, then the lock is free and available for the next reader or writer thread that wants it.

Here is what this class looks like (some method overloads are not shown).

```
public class ReaderWriterLockSlim : IDisposable {
   public ReaderWriterLockSlim(LockRecursionPolicy recursionPolicy);
   public void Dispose();

   public void    EnterReadLock();
   public Boolean TryEnterReadLock(Int32 millisecondsTimeout);
   public void    ExitReadLock();

   public void    EnterWriteLock();
   public Boolean TryEnterWriteLock(Int32 millisecondsTimeout);
   public void    ExitWriteLock();

   // Most applications will never query any of these properties
   public Boolean IsReadLockHeld             { get; }
   public Boolean IsWriteLockHeld            { get; }
   public Int32   CurrentReadCount           { get; }
   public Int32   RecursiveReadCount         { get; }
   public Int32   RecursiveWriteCount        { get; }
   public Int32   WaitingReadCount           { get; }
   public Int32   WaitingWriteCount          { get; }
```

```
   public LockRecursionPolicy RecursionPolicy { get; }
   // Members related to upgrading from a reader to a writer not shown
}
```

Here is some code that demonstrates the use of this construct.

```
internal sealed class Transaction : IDisposable {
   private readonly ReaderWriterLockSlim m_lock =
      new ReaderWriterLockSlim(LockRecursionPolicy.NoRecursion);
   private DateTime m_timeOfLastTrans;

   public void PerformTransaction() {
      m_lock.EnterWriteLock();
      // This code has exclusive access to the data...
      m_timeOfLastTrans = DateTime.Now;
      m_lock.ExitWriteLock();
   }

   public DateTime LastTransaction {
      get {
         m_lock.EnterReadLock();
         // This code has shared access to the data...
         DateTime temp = m_timeOfLastTrans;
         m_lock.ExitReadLock();
         return temp;
      }
   }

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

There are a few concepts related to this construct that deserve special mention. First, `ReaderWriterLockSlim`'s constructor allows you to pass in a `LockRecursionPolicy` flag, which is defined as follows.

```
public enum LockRecursionPolicy { NoRecursion, SupportsRecursion }
```

If you pass the `SupportsRecursion` flag, then the lock will add thread ownership and recursion behaviors to the lock. As discussed earlier in this chapter, these behaviors negatively affect the lock's performance, so I recommend that you always pass `LockRecursionPolicy.NoRecursion` to the constructor (as I've done). For a reader-writer lock, supporting thread ownership and recursion is phenomenally expensive, because the lock must keep track of all the reader threads that it has let into the lock and keep a separate recursion count for each reader thread. In fact, to maintain all this information in a thread-safe way, the `ReaderWriterLockSlim` internally uses a mutually exclusive spinlock! No, I'm not kidding.

The `ReaderWriterLockSlim` class offers additional methods (not shown earlier) that allow a reading thread to upgrade itself to a writer thread. Later, the thread can downgrade itself to a reader thread. The thinking here is that a thread could start reading the data and based on the data's contents, the thread might want to modify the data. To do this, the thread would upgrade itself from a reader to a writer. Having the lock support this behavior deteriorates the lock's performance, and I don't think that this is a useful feature at all. Here's why: a thread can't just turn itself from a reader

into a writer. Other threads may be reading, too, and these threads will have to exit the lock completely before the thread trying to upgrade is allowed to become a writer. This is the same as having the reader thread exit the lock and then immediately acquire it for writing.

Note The FCL also ships a `ReaderWriterLock` construct, which was introduced in the Microsoft .NET Framework 1.0. This construct had so many problems that Microsoft introduced the `ReaderWriterLockSlim` construct in .NET Framework 3.5. The team didn't improve the `ReaderWriterLock` construct for fear of losing compatibility with applications that were using it. Here are the problems with the `ReaderWriterLock`. Even without thread contention, it is very slow. There is no way to opt out of the thread ownership and recursion behaviors, making the lock even slower. It favors reader threads over writer threads, and therefore, writer threads can get queued up and are rarely serviced, which results in denial of service problems.

The OneManyLock Class

I have created my own reader-writer construct that is faster than the FCL's `ReaderWriterLockSlim` class.[4] My class is called `OneManyLock` because it allows access to either one writer thread or many reader threads. The class basically looks like this.

```
public sealed class OneManyLock : IDisposable {
   public OneManyLock();
   public void Dispose();

   public void Enter(Boolean exclusive);
   public void Leave();
}
```

Now I'd like to give you a sense of how it works. Internally, the class has an `Int32` field for the state of the lock, a `Semaphore` object that reader threads block on, and an `AutoResetEvent` object that writer threads block on. The `Int64` state field is divided into five subfields as follows:

- Four bits represent the state of the lock itself. The possibilities are 0=Free, 1=OwnedByWriter, 2=OwnedByReaders, 3=OwnedByReadersAndWriterPending, and 4=ReservedForWriter. The other values are not used.
- Twenty bits (a number from 0 to 1,048,575) represent the number of reader threads reading (RR) that the lock has currently allowed in.
- Twenty bits represent the number of reader threads waiting (RW) to get into the lock. These threads block on the auto-reset event object.

[4] The code is inside the Ch30-1-HybridThreadSync.cs file that is part of the code that accompanies this book. You can download this code from *http://Wintellect.com/Books*.

- Twenty bits represent the number of writer threads waiting (WW) to get into the lock. These threads block on the other semaphore object.

Now, because all the information about the lock fits in a single `Int64` field, I can manipulate this field by using the methods of the `Interlocked` class so the lock is incredibly fast and causes a thread to block only when there is contention.

Here's what happens when a thread enters the lock for shared access.

- If the lock is Free: Set state to OwnedByReaders, RR=1, Return.
- If the lock is OwnedByReaders: RR++, Return.
- Else: RW++, Block reader thread. When the thread wakes, loop around and try again.

Here's what happens when a thread that has shared access leaves the lock.

- RR--
- If RR > 0: Return
- If WW > 0: Set state to ReservedForWriter, WW--, Release 1 blocked writer thread, Return
- If RW == 0 && WW == 0: Set state to Free, Return

Here's what happens when a thread enters the lock for exclusive access:

- If the lock is Free: Set state to OwnedByWriter, Return.
- If the lock is ReservedForWriter: Set state to OwnedByWriter, Return.
- If the lock is OwnedByWriter: WW++, Block writer thread. When thread wakes, loop around and try again.
- Else: Set state to OwnedByReadersAndWriterPending, WW++, Block writer thread. When thread wakes, loop around and try again.

Here's what happens when a thread that has exclusive access leaves the lock:

- If WW == 0 && RW == 0: Set state to Free, Return
- If WW > 0: Set state to ReservedForWriter, WW--, Release 1 blocked writer thread, Return
- If WW == 0 && RW > 0: Set state to Free , RW=0, Wake all blocked reader threads, Return

Let's say that there is currently one thread reading from the lock and another thread wants to enter the lock for writing. The writer thread will first check to see if the lock is Free, and because it is not, the thread will advance to perform the next check. However, at this point, the reader thread could leave the lock, and seeing that RR and WW are both 0, the thread could set the lock's state to Free. This is a problem because the writer thread has already performed this test and moved on. Basically what happened is that the reader thread changed the state that the writer thread was accessing behind its back. I needed to solve this problem so that the lock would function correctly.

To solve the problem, all of these bit manipulations are performed using the technique I showed in the "The Interlocked Anything Pattern" section from Chapter 29. If you recall, this pattern lets you turn any operation into a thread-safe atomic operation. This is what allows this lock to be so fast and have less state in it than other reader-writer locks. When I run performance tests comparing my `OneManyLock` against the FCL's `ReaderWriterLockSlim` and `ReaderWriterLock` classes, I get the following results.

```
Incrementing x in OneManyLock: 330              Fastest
Incrementing x in ReaderWriterLockSlim: 554     ~1.7x slower
Incrementing x in ReaderWriterLock: 984         ~3x slower
```

Of course, because all reader-writer locks perform more logic than a mutually exclusive lock, their performance can be slightly worse. However, you have to weigh this against the fact that a reader-writer lock allows multiple readers into the lock simultaneously.

Before leaving this section, I'll also mention that my Power Threading library (downloadable for free from *http://Wintellect.com/PowerThreading.aspx*) offers a slightly different version of this lock, called `OneManyResourceLock`. This lock and others in the library offer many additional features, such as deadlock detection, the ability to turn on lock ownership and recursion (albeit at a performance cost), a unified programming model for all locks, and the ability to observe the run-time behavior of the locks. For observing behavior, you can see the maximum amount of time that a thread ever waited to acquire a lock, and you can see the minimum and maximum amount of time that a lock was held.

The `CountdownEvent` Class

The next construct is `System.Threading.CountdownEvent`. Internally, this construct uses a `ManualResetEventSlim` object. This construct blocks a thread until its internal counter reaches 0. In a way, this construct's behavior is the opposite of that of a `Semaphore` (which blocks threads while its count is 0). Here is what this class looks like (some method overloads are not shown).

```
public class CountdownEvent : IDisposable {
   public CountdownEvent(Int32 initialCount);
   public void Dispose();
   public void Reset(Int32 count);                // Set CurrentCount to count
   public void AddCount(Int32 signalCount);       // Increments CurrentCount by signalCount
   public Boolean TryAddCount(Int32 signalCount); // Increments CurrentCount by signalCount
   public Boolean Signal(Int32 signalCount);      // Decrements CurrentCount by signameCount
   public Boolean Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken);

   public Int32      CurrentCount { get; }
   public Boolean    IsSet        { get; }        // true if CurrentCount is 0
   public WaitHandle WaitHandle   { get; }
}
```

After a `CountdownEvent`'s `CurrentCount` reaches 0, it cannot be changed. The `AddCount` method throws `InvalidOperationException` when `CurrentCount` is 0, whereas the `TryAddCount` method simply returns `false` if `CurrentCount` is 0.

The Barrier Class

The System.Threading.Barrier construct is designed to solve a very rare problem, so it is unlikely that you will have a use for it. Barrier is used to control a set of threads that are working together in parallel so that they can step through phases of the algorithm together. Perhaps an example is in order: when the CLR is using the server version of its garbage collector (GC), the GC algorithm creates one thread per core. These threads walk up different application threads' stacks, concurrently marking objects in the heap. As each thread completes its portion of the work, it must stop waiting for the other threads to complete their portion of the work. After all threads have marked the objects, then the threads can compact different portions of the heap concurrently. As each thread finishes compacting its portion of the heap, the thread must block waiting for the other threads. After all the threads have finished compacting their portion of the heap, then all the threads walk up the application's threads' stacks, fixing up roots to refer to the new location of the compacted object. Only after all the threads have completed this work is the garbage collector considered complete and the application's threads can be resumed.

This scenario is easily solved using the Barrier class, which looks like this (some method overloads are not shown).

```
public class Barrier : IDisposable {
   public Barrier(Int32 participantCount, Action<Barrier> postPhaseAction);
   public void Dispose();
   public Int64 AddParticipants(Int32 participantCount);   // Adds participants
   public void RemoveParticipants(Int32 participantCount); // Subtracts participants
   public Boolean SignalAndWait(Int32 millisecondsTimeout, CancellationToken
     cancellationToken);

   public Int64 CurrentPhaseNumber    { get; }  // Indicates phase in process (starts at 0)
   public Int32 ParticipantCount      { get; }  // Number of participants
   public Int32 ParticipantsRemaining { get; }  // # of threads needing to call
     SignalAndWait
}
```

When you construct a Barrier, you tell it how many threads are participating in the work, and you can also pass an Action<Barrier> delegate referring to code that will be invoked whenever all participants complete a phase of the work. You can dynamically add and remove participating threads from the Barrier by calling the AddParticipant and RemoveParticipant methods but, in practice, this is rarely done. As each thread completes its phase of the work, it should call SignalAndWait, which tells the Barrier that the thread is done and the Barrier blocks the thread (using a ManualResetEventSlim). After all participants call SignalAndWait, the Barrier invokes the delegate (using the last thread that called SignalAndWait) and then unblocks all the waiting threads so they can begin the next phase.

Thread Synchronization Construct Summary

My recommendation always is to avoid writing code that blocks any threads. When performing asynchronous compute or I/O operations, hand the data off from thread to thread in such a way to avoid the chance that multiple threads could access the data simultaneously. If you are unable to fully

accomplish this, then try to use the `Volatile` and `Interlocked` methods because they are fast and they also never block a thread. Unfortunately, these methods manipulate only simple types, but you can perform rich operations on these types as described in the "The Interlocked Anything Pattern" section in Chapter 29.

There are two main reasons why you would consider blocking threads:

- **The programming model is simplified** By blocking a thread, you are sacrificing some resources and performance so that you can write your application code sequentially without using callback methods. But C#'s async methods feature gives you a simplified programming model without blocking threads.
- **A thread has a dedicated purpose** Some threads must be used for specific tasks. The best example is an application's primary thread. If an application's primary thread doesn't block, then it will eventually return and the whole process will terminate. Another example is an application's GUI thread or threads. Windows requires that a window or control always be manipulated by the thread that created it, so we sometimes write code that blocks a GUI thread until some other operation is done, and then the GUI thread updates any windows and controls as needed. Of course, blocking the GUI thread hangs the application and provides a bad end-user experience.

To avoid blocking threads, don't mentally assign a label to your threads. For example, don't create a spell-checking thread, a grammar-checking thread, a thread that handles this particular client request, and so on. The moment you assign a label to a thread, you have also said to yourself that the thread can't do anything else. But threads are too expensive a resource to have them dedicated to a particular purpose. Instead, you should use the thread pool to rent threads for short periods of time. So, a thread pool thread starts out spell checking, then it changes to grammar checking, and then it changes again to perform work on behalf of a client request, and so on.

If, in spite of this discussion, you decide to block threads, then use the kernel object constructs if you want to synchronize threads that are running in different AppDomains or processes. To atomically manipulate state via a set of operations, use the `Monitor` class with a `private` field.[5] Alternatively, you could use a reader-writer lock instead of `Monitor`. Reader-writer locks are generally slower than `Monitor`, but they allow multiple reader threads to execute concurrently, which improves overall performance and minimizes the chance of blocking threads.

In addition, avoid using recursive locks (especially recursive reader-writer locks) because they hurt performance. However, `Monitor` is recursive and its performance is very good.[6] Also, avoid releasing a lock in a `finally` block because entering and leaving exception-handling blocks incurs a performance hit, and if an exception is thrown while mutating state, then the state is corrupted, and other threads that manipulate it will experience unpredictable behavior and security bugs.

[5] You could use a `SpinLock` instead of `Monitor` because `SpinLocks` are slightly faster. But a `SpinLock` is potentially dangerous because it can waste CPU time and, in my opinion, it is not sufficiently faster than `Monitor` to justify its use.

[6] This is partially because `Monitor` is actually implemented in native code, not managed code.

Of course, if you do write code that holds a lock, your code should not hold the lock for a long time, because this increases the likelihood of threads blocking. In the "Asynchronous Synchronization" section later in this chapter, I will show a technique that uses collection classes as a way to avoid holding a lock for a long time.

Finally, for compute-bound work, you can use tasks (discussed in Chapter 27) to avoid a lot of the thread synchronization constructs. In particular, I love that each task can have one or more continue-with tasks associated with it that execute via some thread pool thread when some operation completes. This is much better than having a thread block waiting for some operation to complete. For I/O-bound work, call the various XxxAsync methods that cause your code to continue running after the I/O operation completes; this is similar to a task's continue-with task.

The Famous Double-Check Locking Technique

There is a famous technique called double-check locking, which is used by developers who want to defer constructing a singleton object until an application requests it (sometimes called *lazy initialization*). If the application never requests the object, it never gets constructed, saving time and memory. A potential problem occurs when multiple threads request the singleton object simultaneously. In this case, some form of thread synchronization must be used to ensure that the singleton object gets constructed just once.

This technique is not famous because it is particularly interesting or useful. It is famous because there has been much written about it. This technique was used heavily in Java, and later it was discovered that Java couldn't guarantee that it would work everywhere. The famous document that describes the problem can be found on this webpage: *www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html*.

Anyway, you'll be happy to know that the CLR supports the double-check locking technique just fine because of its memory model and volatile field access (described in Chapter 29). Here is code that demonstrates how to implement the double-check locking technique in C#.

```
internal sealed class Singleton {
   // s_lock is required for thread safety and having this object assumes that creating
   // the singleton object is more expensive than creating a System.Object object and that
   // creating the singleton object may not be necessary at all. Otherwise, it is more
   // efficient and easier to just create the singleton object in a class constructor
   private static readonly Object s_lock = new Object();

   // This field will refer to the one Singleton object
   private static Singleton s_value = null;

   // Private constructor prevents any code outside this class from creating an instance
   private Singleton() {
      // Code to initialize the one Singleton object goes here...
   }

   // Public, static method that returns the Singleton object (creating it if necessary)
   public static Singleton GetSingleton() {
```

```
      // If the Singleton was already created, just return it (this is fast)
      if (s_value != null) return s_value;

      Monitor.Enter(s_lock);  // Not created, let 1 thread create it
      if (s_value == null) {
         // Still not created, create it
         Singleton temp = new Singleton();

         // Save the reference in s_value (see discussion for details)
         Volatile.Write(ref s_value, temp);
      }
      Monitor.Exit(s_lock);

      // Return a reference to the one Singleton object
      return s_value;
   }
}
```

The idea behind the double-check locking technique is that a call to the `GetSingleton` method quickly checks the `s_value` field to see if the object has already been created, and if it has, the method returns a reference to it. The beautiful thing here is that no thread synchronization is required after the object has been constructed; the application will run very fast. On the other hand, if the first thread that calls the `GetSingleton` method sees that the object hasn't been created, it takes a thread synchronization lock to ensure that only one thread constructs the single object. This means that a performance hit occurs only the first time a thread queries the singleton object.

Now, let me explain why this pattern didn't work in Java. The Java Virtual Machine read the value of `s_value` into a CPU register at the beginning of the `GetSingleton` method and then just queried the register when evaluating the second `if` statement, causing the second `if` statement to always evaluate to `true`, and multiple threads ended up creating `Singleton` objects. Of course, this happened only if multiple threads called `GetSingleton` at exactly the same time, which in most applications is very unlikely. This is why it went undetected in Java for so long.

In the CLR, calling any lock method is a full memory fence, and any variable writes you have before the fence must complete before the fence and any variable reads after the fence must start after it. For the `GetSingleton` method, this means that the `s_value` field must be reread after the call to `Monitor.Enter`; it cannot be cached in a register across this method call.

Inside `GetSingleton`, you see the call to `Volatile.Write`. Here's the problem that this is solving. Let's say that what you had inside the second `if` statement was the following line of code.

```
s_value = new Singleton();  // This is what you'd ideally like to write
```

You would expect the compiler to produce code that allocates the memory for a `Singleton`, calls the constructor to initialize the fields, and then assigns the reference into the `s_value` field. Making a value visible to other threads is called publishing. But the compiler could do this instead: allocate memory for the `Singleton`, publish (assign) the reference into `s_value`, and then call the constructor. From a single thread's perspective, changing the order like this has no impact. But what if, after publishing the reference into `s_value` and before calling the constructor, another thread calls the `GetSingleton` method? This thread will see that `s_value` is not `null` and start to use the `Singleton`

object, but its constructor has not finished executing yet! This can be a very hard bug to track down, especially because it is all due to timing.

The call to `Volatile.Write` fixes this problem. It ensures that the reference in `temp` can be published into `s_value` only after the constructor has finished executing. Another way to solve this problem would be to mark the `s_value` field with C#'s `volatile` keyword. This makes the write to `s_value` volatile, and again, the constructor has to finish running before the write can happen. Unfortunately, this makes all reads volatile, too, and because there is no need for this, you are hurting your performance with no benefit.

In the beginning of this section, I mentioned that the double-check locking technique is not that interesting. In my opinion, developers think it is cool, and they use it far more often than they should. In most scenarios, this technique actually hurts efficiency. Here is a much simpler version of the `Singleton` class that behaves the same as the previous version. This version does not use the double-check locking technique.

```
internal sealed class Singleton {
   private static Singleton s_value = new Singleton();

   // Private constructor prevents any code outside this class from creating an instance
   private Singleton() {
      // Code to initialize the one Singleton object goes here...
   }

   // Public, static method that returns the Singleton object (creating it if necessary)
   public static Singleton GetSingleton() { return s_value; }
}
```

Because the CLR automatically calls a type's class constructor the first time code attempts to access a member of the class, the first time a thread queries `Singleton`'s `GetSingleton` method, the CLR will automatically call the class constructor, which creates an instance of the object. Furthermore, the CLR already ensures that calls to a class constructor are thread safe. I explained all of this in Chapter 8. The one downside of this approach is that the type constructor is called when any member of a class is first accessed. If the `Singleton` type defined some other static members, then the `Singleton` object would be created when any one of them was accessed. Some people work around this problem by defining nested classes.

Let me show you a third way of producing a single `Singleton` object.

```
internal sealed class Singleton {
   private static Singleton s_value = null;

   // Private constructor prevents any code outside this class from creating an instance
   private Singleton() {
      // Code to initialize the one Singleton object goes here...
   }

   // Public, static method that returns the Singleton object (creating it if necessary)
   public static Singleton GetSingleton() {
      if (s_value != null) return s_value;
```

```
      // Create a new Singleton and root it if another thread didn't do it first
      Singleton temp = new Singleton();
      Interlocked.CompareExchange(ref s_value, temp, null);

      // If this thread lost, then the second Singleton object gets GC'd

      return s_value; // Return reference to the single object
   }
}
```

If multiple threads call `GetSingleton` simultaneously, then this version might create two (or more) `Singleton` objects. However, the call to `Interlocked.CompareExchange` ensures that only one of the references is ever published into the `s_value` field. Any object not rooted by this field will be garbage collected later on. Because, in most applications, it is unlikely that multiple threads will call `GetSingleton` at the same time, it is unlikely that more than one `Singleton` object will ever be created.

Now it might upset you that multiple `Singleton` objects could be created, but there are many benefits to this code. First, it is very fast. Second, it never blocks a thread; if a thread pool thread is blocked on a `Monitor` or any other kernel-mode thread synchronization construct, then the thread pool creates another thread to keep the CPUs saturated with work. So now, more memory is allocated and initialized and all the DLLs get a thread attach notification. With `CompareExchange`, this can never happen. Of course, you can use this technique only when the constructor has no side effects.

The FCL offers two types that encapsulate the patterns described in this section. The generic `System.Lazy` class looks like this (some methods are not shown).

```
public class Lazy<T> {
   public Lazy(Func<T> valueFactory, LazyThreadSafetyMode mode);
   public Boolean IsValueCreated { get; }
   public T Value { get; }
}
```

This code demonstrates how it works.

```
public static void Main() {
   // Create a lazy-initialization wrapper around getting the DateTime
   Lazy<String> s = new Lazy<String>(() => DateTime.Now.ToLongTimeString(), true);

   Console.WriteLine(s.IsValueCreated);   // Returns false because Value not queried yet
   Console.WriteLine(s.Value);            // The delegate is invoked now
   Console.WriteLine(s.IsValueCreated);   // Returns true because Value was queried
   Thread.Sleep(10000);                   // Wait 10 seconds and display the time again
   Console.WriteLine(s.Value);            // The delegate is NOT invoked now; same result
}
```

When I run this, I get the following output.

```
False
2:40:42 PM
True
2:40:42 PM    ß Notice that the time did not change 10 seconds later
```

The preceding code constructed an instance of the Lazy class and passed one of the LazyThreadSafetyMode flags into it. Here is what these flags look like and what they mean.

```
public enum LazyThreadSafetyMode {
   None,                     // No thread-safety support at all (good for GUI apps)
   ExecutionAndPublication   // Uses the double-check locking technique
   PublicationOnly,          // Uses the Interlocked.CompareExchange technique
}
```

In some memory-constrained scenarios, you might not even want to create an instance of the Lazy class. Instead, you can call static methods of the System.Threading.LazyInitializer class. The class looks like this.

```
public static class LazyInitializer {
   // These two methods use Interlocked.CompareExchange internally:
   public static T EnsureInitialized<T>(ref T target) where T: class;
   public static T EnsureInitialized<T>(ref T target, Func<T> valueFactory) where T: class;

   // These two methods pass the syncLock to Monitor's Enter and Exit methods internally
   public static T EnsureInitialized<T>(ref T target, ref Boolean initialized,
      ref Object syncLock);
   public static T EnsureInitialized<T>(ref T target, ref Boolean initialized,
      ref Object syncLock, Func<T> valueFactory);
}
```

Also, being able to explicitly specify a synchronization object to the EnsureInitialized method's syncLock parameter allows multiple initialization functions and fields to be protected by the same lock.

Here is an example using a method from this class.

```
public static void Main() {
   String name = null;
   // Because name is null, the delegate runs and initializes name
   LazyInitializer.EnsureInitialized(ref name, () => "Jeffrey");
   Console.WriteLine(name);   // Displays "Jeffrey"

   // Because name is not null, the delegate does not run; name doesn't change
   LazyInitializer.EnsureInitialized(ref name, () => "Richter");
   Console.WriteLine(name);   // Also displays "Jeffrey"
}
```

The Condition Variable Pattern

Let's say that a thread wants to execute some code when a complex condition is true. One option would be to let the thread spin continuously, repeatedly testing the condition. But this wastes CPU time, and it is also not possible to atomically test multiple variables that are making up the complex condition. Fortunately, there is a pattern that allows threads to efficiently synchronize their operations based on a complex condition.

This pattern is called the *condition variable pattern*, and we use it via the following methods defined inside the `Monitor` class.

```
public static class Monitor {
   public static Boolean Wait(Object obj);
   public static Boolean Wait(Object obj, Int32 millisecondsTimeout);

   public static void Pulse(Object obj);
   public static void PulseAll(Object obj);
}
```

Here is what the pattern looks like.

```
internal sealed class ConditionVariablePattern {
   private readonly Object m_lock = new Object();
   private Boolean m_condition = false;

   public void Thread1() {
      Monitor.Enter(m_lock);        // Acquire a mutual-exclusive lock

      // While under the lock, test the complex condition "atomically"
      while (!m_condition) {
         // If condition is not met, wait for another thread to change the condition
         Monitor.Wait(m_lock);      // Temporarily release lock so other threads can get it
      }

      // The condition was met, process the data...

      Monitor.Exit(m_lock);         // Permanently release lock
   }

   public void Thread2() {
      Monitor.Enter(m_lock);        // Acquire a mutual-exclusive lock

      // Process data and modify the condition...
      m_condition = true;

      // Monitor.Pulse(m_lock);     // Wakes one waiter AFTER lock is released
      Monitor.PulseAll(m_lock);     // Wakes all waiters AFTER lock is released

      Monitor.Exit(m_lock);         // Release lock
   }
}
```

In this code, the thread executing the `Thread1` method enters a mutual-exclusive lock and then tests a condition. Here, I am just checking a `Boolean` field, but this condition can be arbitrarily complex. For example, you could check to see if it is a Tuesday in March and if a certain collection object has 10 elements in it. If the condition is false, then you want the thread to spin on the condition, but spinning wastes CPU time, so instead, the thread calls `Wait`. `Wait` releases the lock so that another thread can get it and blocks the calling thread.

The Thread2 method shows code that the second thread executes. It calls Enter to take ownership of the lock, processes some data, which results in changing the state of the condition, and then calls Pulse or PulseAll, which will unblock a thread from its Wait call. Pulse unblocks the longest waiting thread (if any), whereas PulseAll unblocks all waiting threads (if any). However, any unblocked threads don't wake up yet. The thread executing Thread2 must call Monitor.Exit, allowing the lock to be owned by another thread. Also, if PulseAll is called, the other threads do not unblock simultaneously. When a thread that called Wait is unblocked, it becomes the owner of the lock, and because it is a mutual-exclusive lock, only one thread at a time can own it. Other threads can get it after an owning thread calls Wait or Exit.

When the thread executing Thread1 wakes, it loops around and tests the condition again. If the condition is still false, then it calls Wait again. If the condition is true, then it processes the data as it likes and ultimately calls Exit, leaving the lock so other threads can get it. The nice thing about this pattern is that it is possible to test several variables making up a complex condition using simple synchronization logic (just one lock), and multiple waiting threads can all unblock without causing any logic failure, although the unblocking threads might waste some CPU time.

Here is an example of a thread-safe queue that can have multiple threads enqueuing and dequeuing items to it. Note that threads attempting to dequeue an item block until an item is available for them to process.

```
internal sealed class SynchronizedQueue<T> {
   private readonly Object m_lock = new Object();
   private readonly Queue<T> m_queue = new Queue<T>();

   public void Enqueue(T item) {
      Monitor.Enter(m_lock);

      // After enqueuing an item, wake up any/all waiters
      m_queue.Enqueue(item);
      Monitor.PulseAll(m_lock);

      Monitor.Exit(m_lock);
   }

   public T Dequeue() {
      Monitor.Enter(m_lock);

      // Loop while the queue is empty (the condition)
      while (m_queue.Count == 0)
         Monitor.Wait(m_lock);

      // Dequeue an item from the queue and return it for processing
      T item = m_queue.Dequeue();
      Monitor.Exit(m_lock);
      return item;
   }
}
```

Asynchronous Synchronization

I'm not terribly fond of any of the thread synchronization constructs that use kernel-mode primitives, because all of these primitives exist to block a thread from running, and threads are just too expensive to create and not have them run. Here is an example that hopefully clarifies the problem.

Imagine a website into which clients make requests. When a client request arrives, a thread pool thread starts processing the client's request. Let's say that this client wants to modify some data in the server in a thread-safe way, so it acquires a reader-writer lock for writing. Let's pretend that this lock is held for a long time. As the lock is held, another client request comes in, so that thread pool creates a new thread for the client request, and then the thread blocks trying to acquire the reader-writer lock for reading. In fact, as more and more client requests come in, the thread pool creates more and more threads. Thus, all these threads are just blocking themselves on the lock. The server is spending all its time creating threads so that they can stop running! This server does not scale well at all.

Then, to make matters worse, when the writer thread releases the lock, all the reader threads unblock simultaneously and get to run, but now there may be lots of threads trying to run on relatively few CPUs, so Windows is context switching between the threads constantly. The result is that the workload is not being processed as quickly as it could because of all the overhead associated with the context switches.

If you look over all the constructs shown in this chapter, many of the problems that these constructs are trying to solve can be much better accomplished using the `Task` class discussed in Chapter 27. Take the `Barrier` class, for example: you could spawn several `Task` objects to work on a phase and then, when all these tasks complete, you could continue with one or more other `Task` objects. Compared to many of the constructs shown in this chapter, tasks have many advantages:

- Tasks use much less memory than threads and they take much less time to create and destroy.
- The thread pool automatically scales the tasks across available CPUs.
- As each task completes a phase, the thread running that task goes back to the thread pool, where it can do other work, if any is available for it.
- The thread pool has a process-global view of tasks and, as such, it can better schedule these tasks, reducing the number of threads in the process and also reducing context switching.

Locks are popular but, when held for a long time, they introduce significant scalability issues. What would really be useful is if we had asynchronous synchronization constructs where your code indicates that it wants a lock. If the thread can't have it, it can just return and perform some other work, rather than blocking indefinitely. Then, when the lock becomes available, your code somehow gets resumed, so it can access the resource that the lock protects. I came up with this idea after trying to solve a big scalability problem for a customer, and I then sold the patent rights to Microsoft. In 2009, the Patent Office issued the patent (Patent #7,603,502).

The SemaphoreSlim class implements this idea via its WaitAsync method. Here is the signature for the most complicated overload of this method.

```
public Task<Boolean> WaitAsync(Int32 millisecondsTimeout, CancellationToken cancellationToken);
```

With this, you can synchronize access to a resource asynchronously (without blocking any thread).

```
private static async Task AccessResourceViaAsyncSynchronization(SemaphoreSlim asyncLock) {
   // TODO: Execute whatever code you want here...

   await asyncLock.WaitAsync();     // Request exclusive access to a resource via its lock
   // When we get here, we know that no other thread is accessing the resource
   // TODO: Access the resource (exclusively)...

   // When done accessing resource, relinquish lock so other code can access the resource
   asyncLock.Release();

   // TODO: Execute whatever code you want here...
}
```

The SemaphoreSlim's WaitAsync method is very useful but, of course, it gives you semaphore semantics. Usually, you'll create the SemaphoreSlim with a maximum count of 1, which gives you mutual-exclusive access to the resource that the SemaphoreSlim protects. So, this is similar to the behavior you get when using Monitor, except that SemaphoreSlim does not offer thread ownership and recursion semantics (which is good).

But, what about reader-writer semantics? Well, the .NET Framework has a class called ConcurrentExclusiveSchedulerPair, which looks like this.

```
public class ConcurrentExclusiveSchedulerPair {
   public ConcurrentExclusiveSchedulerPair();

   public TaskScheduler ExclusiveScheduler  { get; }
   public TaskScheduler ConcurrentScheduler { get; }

   // Other methods not shown...
}
```

An instance of this class comes with two TaskScheduler objects that work together to provide reader/writer semantics when scheduling tasks. Any tasks scheduled by using ExclusiveScheduler will execute one at a time, as long as no tasks are running that were scheduled using the ConcurrentScheduler. And, any tasks scheduled using the ConcurrentScheduler can all run simultaneously, as long as no tasks are running that were scheduled by using the ExclusiveScheduler. Here is some code that demonstrates the use of this class.

```
private static void ConcurrentExclusiveSchedulerDemo() {
   var cesp = new ConcurrentExclusiveSchedulerPair();
   var tfExclusive = new TaskFactory(cesp.ExclusiveScheduler);
   var tfConcurrent = new TaskFactory(cesp.ConcurrentScheduler);
```

```
   for (Int32 operation = 0; operation < 5; operation++) {
      var exclusive = operation < 2; // For demo, I make 2 exclusive & 3 concurrent

      (exclusive ? tfExclusive : tfConcurrent).StartNew(() => {
         Console.WriteLine("{0} access", exclusive ? "exclusive" : "concurrent");
         // TODO: Do exclusive write or concurrent read computation here...
      });
   }
}
```

Unfortunately, the .NET Framework doesn't come with an asynchronous lock offering reader-writer semantics. However, I have built such a class, which I call `AsyncOneManyLock`. You use it the same way that you'd use a `SemaphoreSlim`. Here is an example.

```
private static async Task AccessResourceViaAsyncSynchronization(AsyncOneManyLock asyncLock) {
   // TODO: Execute whatever code you want here...

   // Pass OneManyMode.Exclusive or OneManyMode.Shared for wanted concurrent access
   await asyncLock.AcquireAsync(OneManyMode.Shared); // Request shared access
   // When we get here, no threads are writing to the resource; other threads may be reading
   // TODO: Read from the resource...

   // When done accessing resource, relinquish lock so other code can access the resource
   asyncLock.Release();

   // TODO: Execute whatever code you want here...
}
```

The following is the code for my `AsyncOneManyLock`.

```
public enum OneManyMode { Exclusive, Shared }

public sealed class AsyncOneManyLock {
   #region Lock code
   private SpinLock m_lock = new SpinLock(true);   // Don't use readonly with a SpinLock
   private void Lock() { Boolean taken = false; m_lock.Enter(ref taken); }
   private void Unlock() { m_lock.Exit(); }
   #endregion

   #region Lock state and helper methods
   private Int32 m_state = 0;
   private Boolean IsFree { get { return m_state == 0; } }
   private Boolean IsOwnedByWriter { get { return m_state == -1; } }
   private Boolean IsOwnedByReaders { get { return m_state > 0; } }
   private Int32 AddReaders(Int32 count) { return m_state += count; }
   private Int32 SubtractReader() { return --m_state; }
   private void MakeWriter() { m_state = -1; }
   private void MakeFree() { m_state = 0; }
   #endregion

   // For the no-contention case to improve performance and reduce memory consumption
   private readonly Task m_noContentionAccessGranter;

   // Each waiting writers wakes up via their own TaskCompletionSource queued here
```

```
private readonly Queue<TaskCompletionSource<Object>> m_qWaitingWriters =
   new Queue<TaskCompletionSource<Object>>();

// All waiting readers wake up by signaling a single TaskCompletionSource
private TaskCompletionSource<Object> m_waitingReadersSignal =
   new TaskCompletionSource<Object>();
private Int32 m_numWaitingReaders = 0;


public AsyncOneManyLock() {
   m_noContentionAccessGranter = Task.FromResult<Object>(null);
}

public Task WaitAsync(OneManyMode mode) {
   Task accressGranter = m_noContentionAccessGranter; // Assume no contention

   Lock();
   switch (mode) {
      case OneManyMode.Exclusive:
         if (IsFree) {
            MakeWriter();  // No contention
         } else {
            // Contention: Queue new writer task & return it so writer waits
            var tcs = new TaskCompletionSource<Object>();
            m_qWaitingWriters.Enqueue(tcs);
            accressGranter = tcs.Task;
         }
         break;

      case OneManyMode.Shared:
         if (IsFree || (IsOwnedByReaders && m_qWaitingWriters.Count == 0)) {
            AddReaders(1); // No contention
         } else { // Contention
            // Contention: Increment waiting readers & return reader task so reader waits
            m_numWaitingReaders++;
            accressGranter = m_waitingReadersSignal.Task.ContinueWith(t => t.Result);
         }
         break;
   }
   Unlock();

   return accressGranter;
}

public void Release() {
   TaskCompletionSource<Object> accessGranter = null;   // Assume no code is released

   Lock();
   if (IsOwnedByWriter) MakeFree(); // The writer left
   else SubtractReader();           // A reader left

   if (IsFree) {
      // If free, wake 1 waiting writer or all waiting readers
```

```
            if (m_qWaitingWriters.Count > 0) {
               MakeWriter();
               accessGranter = m_qWaitingWriters.Dequeue();
            } else if (m_numWaitingReaders > 0) {
               AddReaders(m_numWaitingReaders);
               m_numWaitingReaders = 0;
               accessGranter = m_waitingReadersSignal;

               // Create a new TCS for future readers that need to wait
               m_waitingReadersSignal = new TaskCompletionSource<Object>();
            }
         }
         Unlock();

         // Wake the writer/reader outside the lock to reduce
         // chance of contention improving performance
         if (accessGranter != null) accessGranter.SetResult(null);
      }
   }
```

As I said, this code never blocks a thread. The reason is because it doesn't use any kernel constructs internally. Now, it does use a `SpinLock` that internally uses user-mode constructs. But, if you recall from the discussion about `SpinLocks` in Chapter 29, a `SpinLock` can only be used when held over sections of code that are guaranteed to execute in a short and finite amount of time. If you examine my `WaitAsync` method, you'll notice that all I do while holding the lock is some integer calculations and comparison and perhaps construct a `TaskCompletionSource` and add it to a queue. This can't take very long at all, so the lock is guaranteed to be held for a very short period of time.

Similarly, if you examine my `Release` method, you'll notice that all I do is some integer calculations, a comparison and perhaps dequeue a `TaskCompletionSource` or possibly construct a `TaskCompletionSource`. Again, this can't take very long either. The end result is that I feel very comfortable using a `SpinLock` to guard access to the `Queue`. Therefore, threads never block when using this lock, which allows me to build responsive and scalable software.

The Concurrent Collection Classes

The FCL ships with four thread-safe collection classes, all of which are in the `System.Collections.Concurrent` namespace: `ConcurrentQueue`, `ConcurrentStack`, `ConcurrentDictionary`, and `ConcurrentBag`. Here is what some of their most commonly used members look like.

```
// Process items in a first-in, first-out order (FIFO)
public class ConcurrentQueue<T> : IProducerConsumerCollection<T>,
   IEnumerable<T>, ICollection, IEnumerable {

   public ConcurrentQueue();
   public void Enqueue(T item);
   public Boolean TryDequeue(out T result);
   public Int32 Count { get; }
   public IEnumerator<T> GetEnumerator();
}
```

```
// Process items in a last-in, first-out order (LIFO)
public class ConcurrentStack<T> : IProducerConsumerCollection<T>,
      IEnumerable<T>, ICollection, IEnumerable {

   public ConcurrentStack();
   public void Push(T item);
   public Boolean TryPop(out T result);
   public Int32 Count { get; }
   public IEnumerator<T> GetEnumerator();
}

// An unordered set of items where duplicates are allowed
public class ConcurrentBag<T> : IProducerConsumerCollection<T>,
   IEnumerable<T>, ICollection, IEnumerable {

   public ConcurrentBag();
   public void Add(T item);
   public Boolean TryTake(out T result);
   public Int32 Count { get; }
   public IEnumerator<T> GetEnumerator();
}

// An unordered set of key/value pairs
public class ConcurrentDictionary<TKey, TValue> : IDictionary<TKey, TValue>,
   ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>,
   IDictionary, ICollection, IEnumerable {

   public ConcurrentDictionary();
   public Boolean TryAdd(TKey key, TValue value);
   public Boolean TryGetValue(TKey key, out TValue value);
   public TValue this[TKey key] { get; set; }
   public Boolean TryUpdate(TKey key, TValue newValue, TValue comparisonValue);
   public Boolean TryRemove(TKey key, out TValue value);
   public TValue AddOrUpdate(TKey key, TValue addValue,
      Func<TKey, TValue> updateValueFactory);
   public TValue GetOrAdd(TKey key, TValue value);
   public Int32 Count { get; }
   public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator();
}
```

All these collection classes are non-blocking. That is, if a thread tries to extract an element when no such element exists, the thread returns immediately; the thread does not block waiting for an element to appear. This is why methods like `TryDequeue`, `TryPop`, `TryTake`, and `TryGetValue` all return `true` if an item was obtained and returns `false`, if not.

These non-blocking collections are not necessarily lock-free. The `ConcurrentDictionary` class uses `Monitor` internally, but the lock is held for a very short time while manipulating the item in the collection. `ConcurrentQueue` and `ConcurrentStack` are lock-free; these both internally use `Interlocked` methods to manipulate the collection. A single `ConcurrentBag` object internally consists of a mini-collection object per thread. When a thread adds an item to the bag, `Interlocked` methods are used to add the item to the calling thread's mini-collection. When a thread tries to extract an element from the bag, the bag checks the calling thread's mini-collection for the item. If the item is there, then an `Interlocked` method is used to extract the item. If the thread's mini-collection doesn't have the

item, then a `Monitor` is taken internally to extract an item from another thread's mini-collection. We say that the thread is stealing the item from another thread.

You'll notice that all the concurrent classes offer a `GetEnumerator` method, which is typically used with C#'s `foreach` statement, but can also be used with Language Integrated Query (LINQ). For the `ConcurrentStack`, `ConcurrentQueue`, and `ConcurrentBag`, the `GetEnumerator` method takes a snapshot of the collection's contents and returns elements from this snapshot; the contents of the actual collection may change while enumerating over the snapshot. `ConcurrentDictionary`'s `GetEnumerator` method does not take a snapshot of its contents, so the contents of the dictionary may change while enumerating over the dictionary; beware of this. Also note that the `Count` property returns the number of elements that are in the collection at the moment you query it. The count may immediately become incorrect if other threads are adding or removing elements from the collection at the same time.

Three of the concurrent collection classes, `ConcurrentStack`, `ConcurrentQueue`, and `ConcurrentBag`, implement the `IProducerConsumerCollection` interface, which is defined as follows.

```
public interface IProducerConsumerCollection<T> : IEnumerable<T>, ICollection, IEnumerable {
   Boolean TryAdd(T item);
   Boolean TryTake(out T item);
   T[] ToArray();
   void CopyTo(T[] array, Int32 index);
}
```

Any class that implements this interface can be turned into a blocking collection, where threads producing (adding) items will block if the collection is full and threads consuming (removing) items will block if the collection is empty. Of course, I'd try to avoid using blocking collections because their purpose in life is to block threads. To turn a non-blocking collection into a blocking collection, you construct a `System.Collections.Concurrent.BlockingCollection` class by passing in a reference to a non-blocking collection to its constructor. The `BlockingCollection` class looks like this (some methods are not shown).

```
public class BlockingCollection<T> : IEnumerable<T>, ICollection, IEnumerable, IDisposable {
   public BlockingCollection(IProducerConsumerCollection<T> collection,
      Int32 boundedCapacity);

   public void Add(T item);
   public Boolean TryAdd(T item, Int32 msTimeout, CancellationToken cancellationToken);
   public void CompleteAdding();

   public T Take();
   public Boolean TryTake(out T item, Int32 msTimeout, CancellationToken cancellationToken);

   public Int32   BoundedCapacity   { get; }
   public Int32   Count             { get; }
   public Boolean IsAddingCompleted { get; }  // true if CompleteAdding is called
   public Boolean IsCompleted       { get; }  // true if IsAddingComplete is true and Count==0

   public IEnumerable<T> GetConsumingEnumerable(CancellationToken cancellationToken);
```

```
   public void CopyTo(T[] array, int index);
   public T[] ToArray();
   public void Dispose();
}
```

When you construct a `BlockingCollection`, the `boundedCapacity` parameter indicates the maximum number of items that you want in the collection. If a thread calls `Add` when the underlying collection has reached its capacity, the producing thread will block. If preferred, the producing thread can call `TryAdd`, passing a timeout (in milliseconds) and/or a `CancellationToken`, so that the thread blocks until the item is added, the timeout expires, or the `CancellationToken` is canceled (see Chapter 27 for more information about the `CancellationToken` class).

The `BlockingCollection` class implements the `IDisposable` interface. When you call `Dispose`, it calls `Dispose` on the underlying collection. It also disposes of two `SemaphoreSlim` objects that the class uses internally to block producers and consumers.

When producers will not be adding any more items into the collection, a producer should call the `CompleteAdding` method. This will signal the consumers that no more items will be produced. Specifically, this causes a `foreach` loop that is using `GetConsumingEnumerable` to terminate. The following example code demonstrates how to set up a producer/consumer scenario and signal completion.

```
public static void Main() {
   var bl = new BlockingCollection<Int32>(new ConcurrentQueue<Int32>());

   // A thread pool thread will do the consuming
   ThreadPool.QueueUserWorkItem(ConsumeItems, bl);

   // Add 5 items to the collection
   for (Int32 item = 0; item < 5; item++) {
      Console.WriteLine("Producing: " + item);
      bl.Add(item);
   }

   // Tell the consuming thread(s) that no more items will be added to the collection
   bl.CompleteAdding();

   Console.ReadLine();  // For testing purposes
}

private static void ConsumeItems(Object o) {
   var bl = (BlockingCollection<Int32>) o;

   // Block until an item shows up, then process it
   foreach (var item in bl.GetConsumingEnumerable()) {
      Console.WriteLine("Consuming: " + item);
   }

   // The collection is empty and no more items are going into it
   Console.WriteLine("All items have been consumed");
}
```

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

```
Producing: 0
Producing: 1
Producing: 2
Producing: 3
Producing: 4
Consuming: 0
Consuming: 1
Consuming: 2
Consuming: 3
Consuming: 4
All items have been consumed
```

If you run this yourself, the `Producing` and `Consuming` lines could be interspersed, but the `All items have been consumed` line will always be last.

The `BlockingCollection` class also has static `AddToAny`, `TryAddToAny`, `TakeFromAny`, and `TryTakeFromAny` methods. All of these methods take a `BlockingCollection<T>[]`, in addition to an item, a timeout, and a `CancellationToken`. The `(Try)AddToAny` methods cycle through all the collections in the array until they find a collection that can accept the item because it is below capacity. The `(Try)TakeFromAny` methods cycle through all the collections in the array until they find a collection to remove an item from.