Threading in C#

The mere mention of multithreading can strike fear in the hearts of some programmers. For others, it fires them up for a good challenge. No matter how you react to the subject, multithreading is an area riddled with mine fields. Unless you show due diligence, a threading bug can jump up and bite you—and bite you in a place where you cannot seem to find it easily. Threading bugs can be among the hardest to find, because they are asynchronous. Threading bugs are hard enough to find on a single-processor machine, but add another processor, and the bugs can become even harder to find. In fact, some threading bugs don't even rear their ugly head until you run your application on a multiprocessor machine, since that's the only way to get true concurrent multithreading. For this reason, I always advise anyone developing a multithreaded application to test, and test often, on a multiprocessor machine. Otherwise, you run the risk of sending your product out the door with lurking threading bugs.

I remember it as if it were a meal ago: At a former employer of mine, we were soon to ship our gold master to the manufacturer and have hundreds of thousands of disks made, and then someone finally happened to test the application on a multiprocessor machine in the lab. Needless to say, a great lesson was learned across the entire team, and a nasty bug was snipped before it got out the door.

Threading in C# and .NET

Even though threading environments have presented many challenges and hurdles over the years, and will continue to do so, the CLR and the .NET base class library mitigate many of these risks and provide a clean model to build upon. It's still true that the greatest challenge of creating high-quality threaded code is that of synchronization. The .NET Framework makes it easier than ever to create new threads or utilize a system-managed pool of threads, and it provides intuitive objects that help you synchronize those threads with each other. However, it's still your duty to make sure you use those objects properly.

Managed threads are virtual threads in the sense that they don't map one-to-one to OS threads. Managed threads do actually run concurrently, but it would be erroneous to assume that the OS thread currently running a particular managed thread's code will only run managed code for that thread only. In fact, an OS thread could run managed code for multiple managed threads in multiple application domains in the current implementation of the CLR. The bottom line is, don't make any assumptions about the correlation between OS threads and managed threads. If you burrow down to the OS thread using the P/Invoke layer to make direct Win32 calls, be sure that you only use that information for debugging purposes and base no program logic on it at all. Otherwise, you'll end up with something that may break as soon as you run it on another CLR implementation.

It would be erroneous to conclude that multithreaded programming is just about creating extra threads to do something that can take a long time to do. Sure, that's part of the puzzle. And when

you create a desktop application, you definitely want to use a threading technique to ensure that the UI stays responsive during a long computational operation, because we all know what impatient users tend to do when desktop applications become unresponsive: They kill them! But it's important to realize that there is much more to the threading puzzle than creating an extra thread to run some random code. That task is actually quite easy in the C# environment, so let's take a look and see how easy it really is.

Starting Threads

As I said, creating a thread is very simple. Take a look at the following example to see what I mean:

```
using System;
using System.Threading;

public class EntryPoint
{
    private static void ThreadFunc() {
        Console.WriteLine( "Hello from new thread {0}!",
                           Thread.CurrentThread.GetHashCode() );
    }

    static void Main() {
        // Create the new thread.
        Thread newThread =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );

        Console.WriteLine( "Main Thread is {0}",
                           Thread.CurrentThread.GetHashCode() );
        Console.WriteLine( "Starting new thread..." );

        // Start the new thread.
        newThread.Start();

        // Wait for new thread to finish.
        newThread.Join();

        Console.WriteLine( "New thread has finished" );
    }
}
```

All you have to do is create a new `System.Thread` object and pass an instance of the `ThreadStart` delegate as the parameter to the constructor. The `ThreadStart` delegate references a method that takes no parameters and returns no parameters. In the previous example, I chose to use the static `ThreadFunc` method as the start of execution for the new thread. I could have just as easily chosen to use any other method visible to the code creating the thread, as long as it neither accepted nor returned parameters. Notice that the code also outputs the hash code from the thread to demonstrate how you identify threads in the managed world. In the unmanaged C++ world, you would use the thread ID obtained via the Win32 API. In the managed world, you instead use the value returned by `GetHashCode`. As long as this thread is alive, it is guaranteed never to collide with any other thread in any application domain of this process. The thread hash code is not globally unique on the entire system. Also, you can see how you can get a reference to the current thread by accessing the static property `Thread.CurrentThread`. Finally, notice the call to the `Join` method on the `newThread` object. In native Win32 code, you normally wait for a thread to finish by waiting on its handle. When the thread finishes running, the operating system signals its handle and the wait completes. The

`Thread.Join` method encapsulates this functionality. In this case, the code waits forever for the thread to finish. `Thread.Join` also provides a few overloads that allow you to specify a timeout period on the wait.

In the managed environment, the `System.Thread` class nicely encapsulates all of the operations that you may perform on a thread. If you have some sort of state data that you must transmit to the new thread so that it has that data available when it starts execution, you can simply create a helper object and initialize the `ThreadStart` delegate to point to an instance method on that object. Yet again, you solve another problem by introducing another level of indirection in the form of a class. Suppose you have a system where you fill multiple queues with tasks, and then at some point you want to create a new thread to process the items in a specific queue that you pass into it. The following code demonstrates one way you can achieve such a goal:

```
using System;
using System.Threading;
using System.Collections;

public class QueueProcessor
{
    public QueueProcessor( Queue theQueue ) {
        this.theQueue = theQueue;
        theThread = new Thread( new ThreadStart(this.ThreadFunc) );
    }

    private Queue theQueue;

    private Thread theThread;
    public Thread TheThread {
        get {
            return theThread;
        }
    }

    public void BeginProcessData() {
        theThread.Start();
    }

    public void EndProcessData() {
        theThread.Join();
    }

    private void ThreadFunc() {
        // ... drain theQueue here.
    }
}

public class EntryPoint
{
    static void Main() {
        Queue queue1 = new Queue();
        Queue queue2 = new Queue();

        // ... operations to fill the queues with data.

        // Process each queue in a separate thread.
        QueueProcessor proc1 = new QueueProcessor( queue1 );
```

```
        proc1.BeginProcessData();

        QueueProcessor proc2 = new QueueProcessor( queue2 );
        proc2.BeginProcessData();

        // ... do some other work in the meantime.

        // Wait for the work to finish.
        proc1.EndProcessData();
        proc2.EndProcessData();
    }
}
```

There are some potential synchronization problems here if anyone were to access the queues after the new threads begin their work. But I'll save synchronization issues until later in the chapter. This solution is a clean one and also loosely follows the typical pattern of asynchronous processing in the .NET Framework. The class adding the extra level of indirection is the `QueueProcessor` class. It cleanly encapsulates the worker thread and exposes a lightweight interface to get the work done. In this example, the main thread waits for the work to finish by calling `EndProcessData`. That method merely calls `Join` on the encapsulated thread. However, had you required some sort of status regarding the completion of the work, the `EndProcessData` method could have returned it to you.

When you create a separate thread, it is subject to the rules of the thread scheduler on the system, just like any other thread. However, sometimes you need to create threads that carry a little more or a little less weight when the scheduler algorithm is deciding which thread to execute next. You can control the priority of a managed thread via the `Thread.Priority` property. You can adjust this value as necessary during execution of the thread. It's actually a rare occurrence that you'll need to adjust this value. All threads start out with the priority of `Normal` from the `ThreadPriority` enumeration.

The IOU Pattern and Asynchronous Method Calls

In the section titled "Asynchronous Method Calls," where I discuss asynchronous I/O and thread pools, you'll see that the `BeginProcessData/EndProcessData` is a common pattern of asynchronous processing used throughout the .NET Framework. The `BeginMethod/EndMethod` pattern of asynchronous programming in the .NET Framework is similar to the IOU pattern described by Allan Vermeulen in his article, "An Asynchronous Design Pattern" (*Dr. Dobb's Journal*, June 1996). In that pattern, a function is called to start the asynchronous operation and in return, the caller is given an "I owe you" (IOU) object. Later, the caller can use that object to retrieve the result of the asynchronous operation. The beauty of this pattern is that it completely decouples the caller wanting to get the asynchronous work done from the mechanism used to actually do the work. This pattern is used extensively in the .NET Framework, and I suggest that you employ it for asynchronous method calls, as it will give your clients a familiar look and feel.

States of a Thread

The states of a managed thread are well defined by the runtime. Although the state transitions may seem confusing at times, they aren't much more confusing than the state transitions of an OS thread. There are other considerations to address in the managed world, so the allowable states and state transitions are naturally more complex. Figure 12-1 shows a state diagram for managed threads.

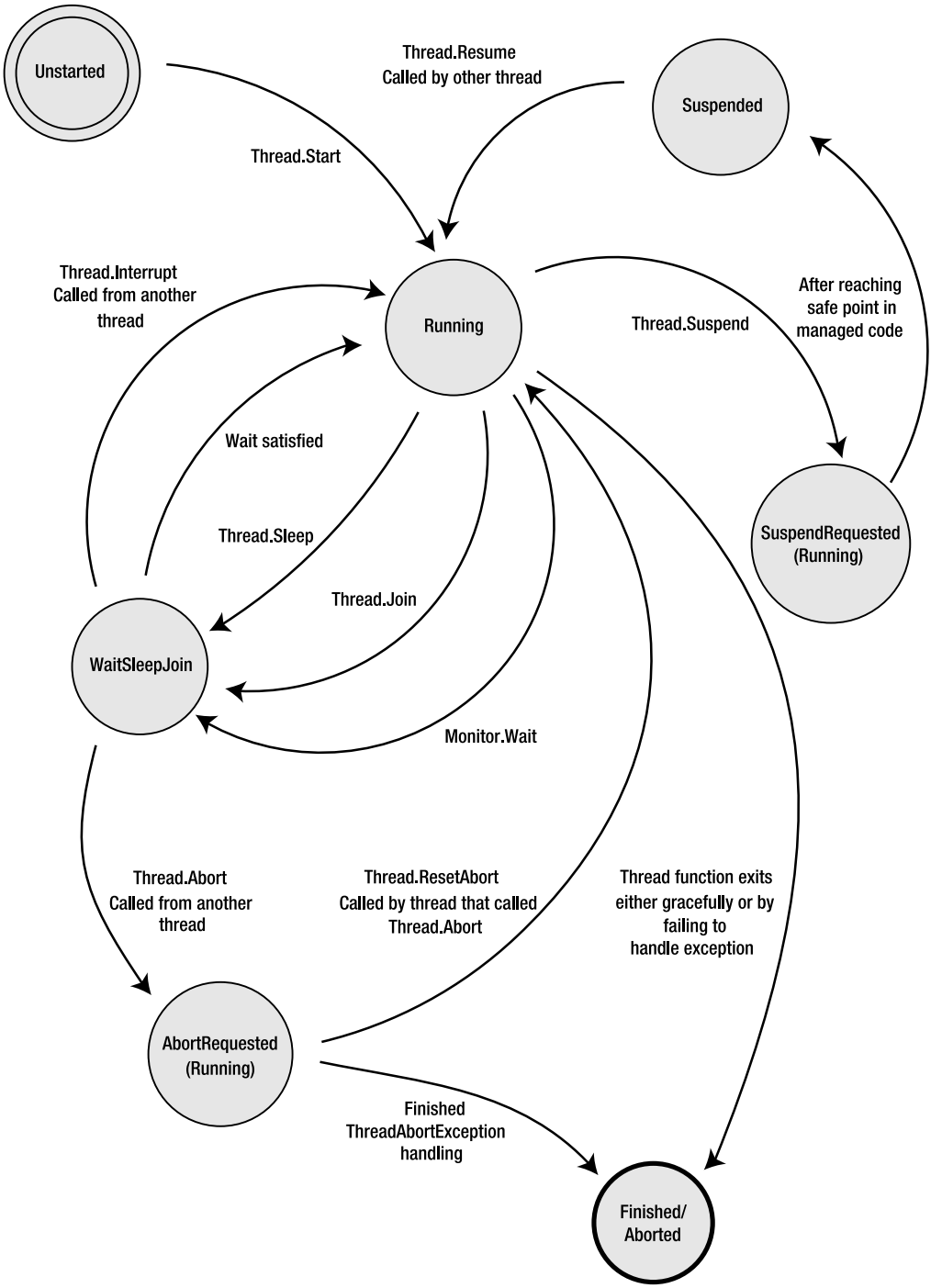

Figure 12-1. *State diagram of managed threads*

The states in the state diagram are based upon the states defined by the CLR for managed threads, as defined in the `ThreadState` enumeration. Every managed thread starts life in the `Unstarted` state. As soon as you call `Start` on the new thread, it enters the `Running` state. OS threads that enter the managed runtime start immediately in the `Running` state, thus bypassing the `Unstarted` state. Notice that there is no way to get back to the `Unstarted` state. The dominant state in the state diagram is the `Running` state. This is the state of the thread when it is executing code normally, including any exception handling and execution of any `finally` blocks. If the main thread method, passed in via an instance of the `ThreadStart` delegate during thread creation, finishes normally, then the thread enters the `Finished` state, as shown in Figure 12-1. Once in this state, the thread is completely dead and will never wake up again. If all of the foreground threads in your process enter the `Finished` state, the process will exit normally.

The three states mentioned previously cover the basics of managed thread state transition, assuming you have a thread that simply executes some code and exits. Once you start to add synchronization constructs in the execution path or wish to control the state of the thread, whether from another thread or the current thread, things become more complicated.

For example, suppose you're writing code for a new thread and you want to put it to sleep for a while. You would call `Thread.Sleep` and provide it a timeout, such as how many milliseconds to sleep. This is similar to how you put an OS thread to sleep. When you call `Sleep`, the thread enters the `WaitSleepJoin` state, where its execution is suspended for the duration of the timeout. Once the sleep expires, the thread reenters the running state.

Synchronization operations can also put the thread into the `WaitSleepJoin` state. As may be obvious by the name of the state, calling `Thread.Join` on another thread in order to wait for it to finish puts the calling thread into the `WaitSleepJoin` state. Calling `Monitor.Wait` also enters the `WaitSleepJoin` state. Now you know the three factors that went into naming the state in the first place. You can use other synchronization methods with a thread, and I'll cover those later in the chapter in the "Synchronizing Work Between Threads" section. As before, once the thread's wait requirements have been met, it reenters the `Running` state and continues execution normally.

It's important to note that any time the thread is sitting in the `WaitSleepJoin` state, it can be forcefully pushed back into the `Running` state when another thread calls `Thread.Interrupt` on the waiting thread. Win32 programmers will recognize that this behavior is similar to alertable wait states in the operating system. Beware that when a thread calls `Thread.Interrupt` on another thread, the interrupted thread receives a thrown `ThreadInterruptedException`. So, even though the interrupted thread reenters the `Running` state, it won't stay there for long unless an appropriate exception-handling frame is in place. Otherwise, the thread will soon enter the `Finished` state once the exception boils its way up to the top of the thread's stack unhandled.

Another way that the thread state can transition out of the `WaitSleepJoin` state is when another thread calls `Thread.Abort` on the current thread. Technically, a thread could call `Abort` on itself. However, I consider that a rare execution flow and have not shown it in Figure 12-1. Once `Thread.Abort` is called, the thread enters the `AbortRequested` state. This state is actually a form of a running state, since the thread is thrown a `ThreadAbortException` and must handle the exception. However, as I explain later on, the managed thread treats this exception in a special way, such that the next state will be the final `Aborted` state unless the thread that called `Thread.Abort` manages to call `Thread.ResetAbort` before that happens. Incidentally, there's nothing to stop the thread that is aborting from calling `ResetAbort`. However, you must refrain from doing such a thing since it could create some ill behavior. For example, if a foreground thread can never be aborted because it keeps resetting the abort, the process will never exit.

■ **Note** Beginning in .NET 2.0, the host has the ability to forcefully kill threads during application domain shutdown by using what's called a *rude thread abort*. In such a situation, it is impossible for the thread to keep itself alive by using `Thread.ResetAbort`.

Finally, a running thread enters the SuspendRequested state after calling Thread.Suspend on itself, or after another thread calls Suspend on it. Very shortly after that, the thread automatically enters the Suspended state. Once a thread enters the SuspendRequested state, there is no way to keep it from eventually entering the Suspended state. Later on, in the section titled "Halting Threads and Waking Sleeping Threads," I discuss why this intermediate state is needed when a thread is suspended. But for now, it's important to realize that the SuspendRequested state is a form of a running state in the sense that it is still executing managed code.

That wraps up the big picture regarding managed-thread state transitions. Be sure to refer to Figure 12-1 throughout the rest of the chapter when reading about topics that affect the state of the thread.

Terminating Threads

When you call Thread.Abort, the thread in question eventually receives a ThreadAbortException. So, naturally, in order to handle this situation gracefully, you must process the ThreadAbortException if there is anything specific you must do when the thread is being aborted. There is also an overload of Abort that accepts an arbitrary object reference, which is then encapsulated in the subsequent ThreadAbortException. This allows the code that is aborting the thread to pass some sort of context information to the ThreadAbortException handler, such as a reason why Abort was called in the first place.

The CLR doesn't deliver a ThreadAbortException unless the thread is running within the managed context. If your thread has called out to a native function via the P/Invoke layer, and that function takes a long time to complete, then a thread abort on that thread is pended until execution returns to managed space.

■ **Note** In .NET 2.0 and later, if a finally block is executing, delivery of a ThreadAbortException is pended until execution leaves the finally block. In .NET 1.x, the abort exception is delivered anyway.

Calling Abort on a thread doesn't forcefully terminate the thread, so if you need to wait until the thread is truly finished executing, you must call Join on that thread to wait until all of the code in the ThreadAbortException exception handler is finished. During such a wait, it is wise to wait with a timeout so that you don't get stuck waiting forever for a thread to finish cleaning up after itself. Even though the code in the exception handler should follow other exception-handler coding guidelines, it's still possible for the handler to take a long time or, gasp, forever to complete its work. Let's take a look at a ThreadAbortException handler and see how this works:

```
using System;
using System.Threading;

public class EntryPoint
{
    private static void ThreadFunc() {
        ulong counter = 0;
        while( true ) {
            try {
                Console.WriteLine( "{0}", counter++ );
            }
            catch( ThreadAbortException ) {
                // Attempt to swallow the exception and continue.
                Console.WriteLine( "Abort!" );
```

```
            }
        }
    }

    static void Main() {
        Thread newThread =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        newThread.Start();
        Thread.Sleep( 2000 );

        // Abort the thread.
        newThread.Abort();

        // Wait for thread to finish.
        newThread.Join();
    }
}
```

From a cursory glance at the code, it would appear that the call to `Join` on the `newThread` instance will block forever. However, that's not what happens. It would appear that since the `ThreadAbortException` is handled within the loop of the thread function, the exception will be swallowed and the loop will continue no matter how many times the main thread attempts to abort the thread. As it turns out, the `ThreadAbortException` thrown via the `Thread.Abort` method is special. When your thread finishes processing the abort exception, the runtime implicitly rethrows it at the end of your exception handler. It's the same as if you had rethrown the exception yourself. Therefore, any outer exception handlers or `finally` blocks will still execute normally. In the example, the call to `Join` won't be waiting forever as initially expected.

There is a way to keep the system from rethrowing the `ThreadAbortException`, by calling the `Thread.ResetAbort` static method. However, the general recommendation is that you only call `ResetAbort` from the thread that called `Abort`. This would require some sort of tricky intrathread communication technique if you wanted to cause this to happen from within the abort handler of the thread being aborted. If you find yourself trying to implement such a technique to abort a thread abort, then maybe it's time to reassess the design of the system in the first place. In other words, bad design alert!

Even though the runtime provides a much cleaner mechanism for aborting threads such that you can inform interested parties when the thread is aborting, you still have to implement a `ThreadAbortException` handler properly.

■ **Note** The fact that `ThreadAbortException` instances can be thrown asynchronously into a random managed thread makes it tricky to create robust exception-safe code. Be sure to read the "Constrained Execution Regions" section in Chapter 7.

Halting Threads and Waking Sleeping Threads

Similar to native threads, there are mechanisms in place for putting a thread to sleep for a defined period of time or actually halting execution until it is explicitly released again. If a thread just wants to suspend itself for a prescribed period of time, it may call the static method `Thread.Sleep`. The only parameter to the `Sleep` method is the number of milliseconds the thread should sleep. When called, this method causes the thread to relinquish the rest of its time slice with the processor and go to sleep. After the time has expired, the thread may be considered for scheduling again. Naturally, the time duration you pass to `Sleep` is reasonably accurate, but not exact. That's because, at

the end of the duration, the thread is not immediately given time on the processor. There could be other, higher-priority threads in the queue before it. Therefore, using `Sleep` to synchronize execution between two threads is strongly discouraged.

■ **Caution** If you find yourself solving synchronization problems by introducing calls to `Sleep` within your code, you're not solving the problems at all. You're merely covering them up even more.

There is even a special value, `Timeout.Infinite`, that you can pass to `Sleep` to make the thread go to sleep forever. You can wake a sleeping thread by interrupting it via the `Thread.Interrupt` instance method. `Interrupt` is similar to `Abort` in that it wakes up the target thread and throws a `ThreadInterruptedException`. Therefore, if your thread function is not equipped to handle the exception, it will percolate all the way up the call stack until the runtime ends the thread's execution. To be safe, you should make your call to `Sleep` within a `try` block and catch the `ThreadInterruptException`. Unlike the `ThreadAbortException`, the `ThreadInterruptException` is not automatically rethrown by the runtime at the end of the exception handler.

■ **Note** Another special parameter value for `Thread.Sleep` is 0. If you pass 0, `Thread.Sleep` will cause the thread to relinquish the rest of its time slice. The thread will then be allowed to run again once the system thread scheduler comes back around to it.

Another way to put a thread to sleep for an indefinite time is via the `Thread.Suspend` instance method. Calling `Suspend` will suspend execution of the thread until it is explicitly resumed. You can resume the thread by calling the `Resume` instance method or `Interrupt`. However, with `Interrupt`, the target thread needs to have a proper exception handler around the `Suspend` call; otherwise, the thread will exit. Technically, calling `Abort` on the thread will resume the thread, but only to send it a `ThreadAbortException` and cause the thread to exit. Keep in mind that any thread with sufficient privileges can call `Suspend` on a thread—even the current thread can call `Suspend`. If the current thread calls `Suspend`, it blocks at that point, waiting for the next `Resume` call.

It's important to note that when you call `Suspend` on a thread, the thread is not suspended immediately in its tracks. Instead, the thread is allowed to execute to what's called a *safe point*. Once it reaches the safe point, the thread is suspended. A safe point is a place in the managed code where it is safe to allow garbage collection. For instance, if the CLR determines it is time to perform a garbage collection, it must suspend all threads temporarily while it performs the collection. However, as you can imagine, if a thread is in the middle of a multi-instruction operation that accesses an object on the heap, and then the GC comes along and moves that object to a different place in system memory, only bad things will happen. For that reason, when the GC suspends threads for collection, it must wait until they all have reached a safe point where it is OK to move things around on the heap. For this reason, the call to `Suspend` allows the thread to reach a safe point before actually suspending it. I also want to stress that you should never use `Suspend` and `Resume` to orchestrate thread synchronization. Of course, the fact that the system allows the thread to continue running until it reaches a safe point is a good enough reason not to rely on this mechanism, but it's also a bad design practice.

Waiting for a Thread to Exit

In this chapter's previous examples, I've used the `Join` method to wait for a specific thread to exit. In fact, that is exactly what it is used for. In an unmanaged Win32 application, you may have been accustomed to waiting for the thread handle to become signaled to indicate the completion of the

thread. The Join method is the same mechanism indeed. The name of the method is suggestive of the fact that you're joining the current thread's execution path to that of the thread you're calling Join on, and you cannot proceed until your joined thread arrives.

Naturally, you'll want to avoid calling Join on the current thread. The effect is similar to calling Suspend from the current thread. The thread is blocked until it is interrupted. Even when a thread is blocked from calling Join, it can be awoken via a call to Interrupt or Abort as described in the previous section.

Sometimes, you'll want to call Join to wait for another thread to complete, but you won't want to get stuck waiting forever. Join offers overloads that allow you to designate the amount of time you're willing to wait. Those overloads return a Boolean value that returns true to indicate that the thread actually terminated, or false to indicate that the timeout expired.

Foreground and Background Threads

When you create a thread in the .NET managed environment, it exists as a foreground thread by default. This means that the managed execution environment, and thus the process, will remain alive as long as the thread is alive. Consider the following code:

```
using System;
using System.Threading;

public class EntryPoint
{
    private static void ThreadFunc1() {
        Thread.Sleep( 5000 );
        Console.WriteLine( "Exiting extra thread" );
    }

    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc1) );

        thread1.Start();

        Console.WriteLine( "Exiting main thread" );
    }
}
```

If you run this code, you'll see that Main exits before the extra thread finishes, as expected. (C++ developers will find that very different from the behavior they're used to, where the process normally terminates once the main routine in the application exits.)

At times, you might want the process to terminate when the main thread finishes, even when there are extra threads in the background. You can accomplish this in the runtime by turning the extra thread into a background thread by setting the Thread.IsBackground property to true. You'll want to consider doing this for threads that do stuff such as listen on a port for network connections, or some other background task such as that. Keep in mind, though, that you always want to make sure that your threads get a proper chance to clean up if they need to before they are shut down. When a background thread is shut down as the process exits, it doesn't receive an exception of any type as it does when someone calls Interrupt or Abort. So, if the thread has persistent data in some sort of half-baked state, shutting down the process will definitely not be good for that persistent data. Therefore, when creating background threads, make sure they are coded so that they can

be terminated rudely at any point without any adverse effects. You can also implement some sort of mechanism to notify the thread that the process is to shut down soon. Creating such a mechanism will prove messy, since the main thread will need to wait a reasonable amount of time after firing the notification for the extra thread to do its cleanup work. At that point, it almost becomes reasonable to turn the thread back into a foreground thread.

Thread-Local Storage

You can create thread-local storage in the managed environment. Depending on your application, it may be necessary for you to have a static field of a class that is unique for each thread that the class is used in. Doing so is trivially easy in the majority of the cases in C#. If you have a static field that must be thread-relative, simply adorn it with the `ThreadStaticAttribute` attribute. Once you do that, the field will be initialized for each thread that accesses it. Under the covers, each thread is given its own thread-relative location to save the value or reference. However, when using references to objects, be careful with your assumptions about object creation. The following code shows a pitfall to avoid:

```
using System;
using System.Threading;

public class TLSClass
{
    public TLSClass() {
        Console.WriteLine( "Creating TLSClass" );
    }
}

public class TLSFieldClass
{
    [ThreadStatic]
    public static TLSClass tlsdata = new TLSClass();
}

public class EntryPoint
{
    private static void ThreadFunc() {
        Console.WriteLine( "Thread {0} starting...",
                           Thread.CurrentThread.GetHashCode() );
        Console.WriteLine( "tlsdata for this thread is \"{0}\"",
                           TLSFieldClass.tlsdata );
        Console.WriteLine( "Thread {0} exiting",
                           Thread.CurrentThread.GetHashCode() );
    }

    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        Thread thread2 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );

        thread1.Start();
        thread2.Start();
    }
}
```

This code creates two threads that access a thread-relative static member of `TLSFieldClass`. To illustrate the trap, I've made that thread-specific slot of type `TLSClass`, and the code attempts to initialize that slot with an initializer in the class definition that simply calls `new` on the default constructor of the class. Now, look how surprising the output is:

```
Thread 3 starting...
Thread 4 starting...
Creating TLSClass
tlsdata for this thread is "TLSClass"
Thread 3 exiting
tlsdata for this thread is ""
Thread 4 exiting
```

■ **Caution** Always remember that ordering of execution in multithreaded programs is never guaranteed unless you employ specific synchronization mechanisms. This output was generated on a single-processor system. If you run the same application on a multiprocessor system, you'll likely see that the output executes in a completely different order. Nevertheless, the purpose of the example does not change.

The important thing to take note of is that the constructor for `TLSClass` was only called once. The constructor was called for the first thread, but not for the second thread. For the second thread, the field is initialized to `null`. Since `tlsdata` is static, its initialization is actually done at the time the static constructor for the `TLSFieldClass` is called. However, static constructors can only be called once per class per application domain. For this reason, you want to avoid assigning thread-relative slots at the point of declaration. That way, they will always be assigned to their default values. For reference types, that means `null`, and for value types, it means the equivalent of setting all of the bits in the value's underlying storage to 0. Then, upon first access to the thread-specific slot, you can test the value for `null` and create an instance as appropriate. Of course, the cleanest way to achieve this is always to access the thread-local slot via a static property.

As an added note, don't think that you can outsmart the compiler by adding a level of indirection, such as assigning the thread-relative slot based on the return value of a static method. You'll find that your static method will only get called once. If the CLR were to "fix" this problem for you, it would undoubtedly be less efficient because it would have to test whether the field is being accessed for the first time and call the initialization code if that is the case. If you think about it, you'll find that task is a lot harder than it sounds, since it will be impossible to do the right thing 100% of the time.

There is another way to use thread-local storage that doesn't involve decorating a static method with an attribute. You can allocate thread-specific storage dynamically by using either of the `Thread.AllocateDataSlot` or `Thread.AllocateNamedDataSlot` methods. You'll want to use these methods if you won't know how many thread-specific slots you'll need to allocate until runtime. Otherwise, it's generally much easier to use the static field method. When you call `AllocateDataSlot`, a new slot is allocated in all threads to hold a reference to an instance of type `System.Object`. The method returns a handle of sorts in the form of a `LocalDataStoreSlot` object instance. You can access this location using the `GetData` and `SetData` methods on the thread. Let's look at a modification of the previous example:

```
using System;
using System.Threading;

public class TLSClass
{
```

```
    static TLSClass() {
        tlsSlot = Thread.AllocateDataSlot();
    }

    public TLSClass() {
        Console.WriteLine( "Creating TLSClass" );
    }

    public static TLSClass TlsSlot {
        get {
            Object obj = Thread.GetData( tlsSlot );
            if( obj == null ) {
                obj = new TLSClass();
                Thread.SetData( tlsSlot, obj );
            }
            return (TLSClass) obj;
        }
    }

    private static LocalDataStoreSlot tlsSlot = null;
}

public class EntryPoint
{
    private static void ThreadFunc() {
        Console.WriteLine( "Thread {0} starting...",
                           Thread.CurrentThread.GetHashCode() );
        Console.WriteLine( "tlsdata for this thread is \"{0}\"",
                           TLSClass.TlsSlot );
        Console.WriteLine( "Thread {0} exiting",
                           Thread.CurrentThread.GetHashCode() );
    }

    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        Thread thread2 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );

        thread1.Start();
        thread2.Start();
    }
}
```

As you can see, using dynamic slots is a little more involved than using the static field method. However, it does provide some extra flexibility. Notice that the slot is allocated in the type initializer, which is the static constructor you see in the example. That way, the slot is allocated for all threads at the point where the runtime initializes the type for use. Notice that I'm testing the slot for `null` in the property accessor of the `TLSClass`. When you allocate the slot using `AllocateDataSlot`, the slot is initialized to `null` for each thread.

You may find it convenient to access your thread-specific storage via a string name rather than with a reference to a `LocalDataStoreSlot` instance. However, you must be careful to use a reasonably unique name so that use of that same name elsewhere in the code won't cause adverse effects. You may consider naming your slot using a string representation of a GUID, so that you can reasonably assume that nobody will attempt to create one with the same name. When you need to access

the slot, you can call `GetNamedDataSlot`, which will simply translate your string into a `LocalDataStoreSlot` instance. I urge you to read the MSDN documentation regarding named thread-local storage slots to get more details.

Most of this will be familiar to those developers who have used thread-local storage in Win32. There is one improvement, though: Because managed TLS slots are implemented in a different way, the limitation on the number of Win32 TLS slots doesn't apply.

How Unmanaged Threads and COM Apartments Fit In

It is possible for unmanaged threads to enter the managed environment from the outside. For example, managed objects can be exposed to native code via the COM interop layer. When the native thread calls through to the object, it enters the managed environment. When this happens, the CLR makes note of that fact, and if it is the first time the unmanaged thread has called into the CLR, it sets up the necessary bookkeeping structures allowing it to run as a managed thread within the managed runtime. As I mentioned before, threads that enter the managed environment this way initially start their managed thread existence in the `Running` state, as shown in Figure 12-1. Once this bookkeeping is set up, then each time the same unmanaged thread enters the runtime, it is associated with the same managed thread.

Just as managed objects can be exposed to the native world as COM objects, COM objects can be exposed to the managed world as managed objects. When a managed thread calls out to a COM object in this way, the runtime relinquishes control over the thread's state until it reenters the managed environment.

Suppose a COM object, written in native C++, calls the `WaitForSingleObject` Win32 API function to wait for a particular synchronization object to become signaled. Then, if a managed thread calls `Thread.Abort` or `Thread.Interrupt` to wake up the thread, the wakeup will be pended until the thread reenters the managed environment. In other words, it will have no effect while the thread is executing unmanaged code. Therefore, you want to be reasonably cognizant of what sorts of synchronization mechanisms are being used by native COM objects that your native code is calling out to.

Finally, if you've ever done an extensive amount of COM development in the past, then you're familiar with the notion of a COM apartment and the proxies and stubs that go along with them.[1] When managed code calls out into COM objects, it is important that the managed code be set up to call the unmanaged COM object through either a single-threaded apartment (STA) or a multi-threaded apartment (MTA). You can set this property on a new managed thread by setting the `Thread.ApartmentState` property. Once the thread makes a COM call, this state gets locked in. In other words, you cannot change it afterwards. You can set the property after the first COM call all you want, but it will have no effect. When you call out to COM objects from managed code, it's best to know the type of apartment the COM objects will run in. That way, you can judiciously choose which type of COM apartment you want your thread to run in. Choosing the wrong type may introduce inefficiencies by forcing calls to go through proxies and stubs. In even worse cases, COM objects may not be callable from other apartment types.

Using `Thread.ApartmentState`, you can control the COM apartment property for new managed threads that you create. But what about the main thread of an application? The fact is that once the main thread of a managed application is running, it's already too late to set the `ApartmentState` property. That's because the managed runtime initializes the main thread to the MTA state as the managed application is initialized. If you need to change the `ApartmentState` of the main thread to STA, the only way to do so is by decorating the `Main` method with the `STAThreadAttribute` attribute.

---

1. For a detailed description of COM apartments and how they work, I suggest you read Don Box's *Essential COM* (Boston, MA: Addison-Wesley Professional, 1997).

Incidentally, you could also decorate it with the `MTAThreadAttribute` attribute, but that would be redundant since that's the CLR's default choice. The following code shows an example of what I'm talking about:

```
public class EntryPoint
{
    [STAThread]
    static void Main() {
    }
}
```

If you've ever worked with Windows Forms applications, especially those generated by the wizards of Visual Studio, you probably have already seen this attribute and wondered what it was all about. By decorating the main UI thread of GUI applications with this attribute, you can integrate native ActiveX controls more easily in the GUI, since those normally run in an STA.

Note that the `ApartmentState` property of a managed thread has no effect on the execution of managed code. And more importantly, when managed objects are consumed by native applications via the COM interop layer, the `ApartmentState` doesn't control what apartment the object appears to live in from the perspective of the native application. From the native side of the fence, all managed objects appear as COM objects that live in the MTA and integrate the Free Threaded Marhsaller (FTM). Also, all threads created in the CLR's thread pool always live in the MTA for the process.

Synchronizing Work Between Threads

Synchronization is arguably the most difficult part of creating multithreaded applications. You can create extra threads to do work all day long without having to worry about synchronization, as long as those threads consume some data at startup that no other thread uses and do some work. Nobody needs to know when they finish or what the results of their operations are. Obviously, it's a rare case that you'll create such a thread. In most cases, you need to communicate with the running thread, wait for it to reach a defined state in the code, or possibly work on the same object or value instances that other threads are working on.

In all of those cases, and more, you must rely upon synchronization techniques to synchronize the threads to avoid race conditions and deadlocks. With race conditions, two threads may need to access the same piece of memory and only one can safely do so at a time. In these cases, you must use a synchronization mechanism that will only allow one thread at a time to access the data and lock out the other thread, making it wait until the first one is done. Multithreaded environments are stochastic in nature, and you never know when the scheduler will take away control from the thread. The classic example is where one thread gets halfway through changing a block of memory, loses control, and then the other thread is given control and starts reading the memory, assuming that it is in a valid state. An example of a deadlock is when two threads are waiting for each other to release a resource. Both threads end up waiting for each other, and since neither one of them can run until the wait is satisfied, they will end up waiting forever.

In all synchronization tasks, you should use the most lightweight sync mechanism that you can get away with and no heavier. For example, if you're trying to share a data block between two threads in the same process and you must gate access between the two, use something such as a `Monitor` lock rather than a `Mutex`. Why? Because a `Mutex` is meant to gate access to a shared resource between processes, and therefore, is a heavyweight OS object that slows down the process when acquiring and releasing the lock. If no interprocess locking is necessary, use the `Monitor` instead. Even more lightweight than the `Monitor` is a set of methods in the `Interlocked` class. These are ideal when you know that the likelihood of actually having to wait a good while when acquiring a lock is low.

■ **Note** Any type of wait on a kernel object—such as waiting on a `Mutex`, `Semaphore`, `EventWaitHanldle`, or any other wait that boils down to waiting on a Win32 kernel object—requires a transition to kernel mode. Transitions to kernel mode are expensive, and you should avoid them if at all possible. For example, if the threads you are synchronizing live in the same process, kernel synchronization objects are probably too heavy. The lightest synchronization technique involves crafty use of the `Threading.Interlocked` class. Its methods are all implemented completely in user mode, thus allowing you to avoid the user-to-kernel mode transition.

When using synchronization objects in a multithreaded environment, you want to hold the lock for as little time as possible. For example, if you acquire a synchronization lock to read a shared structure instance, and code within the method that acquires the lock uses that instance of the structure for some purpose, it's best to make a local copy of the structure on the stack and then release the lock immediately, unless it is logically impossible. That way, you don't tie up other threads in the system that need to access the guarded variable.

When you need to synchronize thread execution, never rely upon methods such as `Thread.Suspend` or `Thread.Resume` to control thread synchronization. If you recall from a previous section in this chapter, calling `Thread.Suspend` doesn't actually suspend the thread immediately. Instead, it must get to a safe point within the managed code before it can suspend execution. And never use `Thread.Sleep` to synchronize threads. `Thread.Sleep` is appropriate when you're doing some sort of polling loop on an entity, such as device hardware that has just been reset and has no way of notifying anyone that it is back online. In that case, you don't want to check the state in a loop repeatedly. Instead, it's much nicer to sleep a little bit between polling, to allow the scheduler to let other threads run. I've said this in a previous section, but I'll say it again because it's so important: If you ever find yourself solving a synchronization bug by introducing a call to `Thread.Sleep` at some seemingly random point in the code, you're not solving the problem at all. Rather, you're hiding it even deeper. Just don't do it!

Lightweight Synchronization with the Interlocked Class

Those of you who come from the unmanaged world of programming against the Win32 API probably already know about the `Interlocked...` family of functions. Thankfully, those functions have been exposed to managed C# developers via static methods on the `Interlocked` class in the `System.Threading` namespace. Sometimes, when running multiple threads, it's necessary to maintain a simple variable—typically, a value, but possibly an object—between the multiple threads. For example, suppose you have some reason to track the number of running threads in a static integer somewhere. When a thread begins, it increments that value, and when it finishes, it decrements that value. Obviously, you must synchronize access to that value somehow, since the scheduler could take away control from one thread and give it to another when the first one is in the process of updating the value. Even worse, the same code could be executing concurrently on a multiprocessor machine. For this task, you can use `Interlocked.Increment` and `Interlocked.Decrement`. These methods are guaranteed to modify the value atomically across all processors in the system. Take a look at the following example:

```
using System;
using System.Threading;

public class EntryPoint
{
    static private int numberThreads = 0;

    static private Random rnd = new Random();
```

```
    private static void RndThreadFunc() {
        // Manage thread count and wait for a
        // random amount of time between 1 and 12
        // seconds.
        Interlocked.Increment( ref numberThreads );
        try {
            int time = rnd.Next( 1000, 12000 );
            Thread.Sleep( time );
        }
        finally {
            Interlocked.Decrement( ref numberThreads );
        }
    }

    private static void RptThreadFunc() {
        while( true ) {
            int threadCount = 0;
            threadCount =
                Interlocked.Exchange( ref numberThreads,
                                      numberThreads );
            Console.WriteLine( "{0} thread(s) alive",
                               threadCount );
            Thread.Sleep( 1000 );
        }
    }

    static void Main() {
        // Start the reporting threads.
        Thread reporter =
            new Thread( new ThreadStart(
                                EntryPoint.RptThreadFunc) );
        reporter.IsBackground = true;
        reporter.Start();

        // Start the threads that wait random time.
        Thread[] rndthreads = new Thread[ 50 ];
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
                new Thread( new ThreadStart(
                                  EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

This little program creates 50 foreground threads that do nothing but wait a random period of time between 1 and 12 seconds. It also creates a background thread that reports how many threads are currently alive. If you look at the `RndThreadFunc` method, which is the thread function that the 50 threads use, you can see it increment and decrement the integer value using the `Interlocked` methods. Notice that I use a `finally` block to ensure that the value gets decremented no matter how the thread exits. You could use the disposable trick with the `using` keyword by wrapping the increment and decrement in a separate class that implements `IDisposable`. That would get rid of the ugly `finally` block. But, in this case, it wouldn't help you at all, since you'd also have to create a reference type to contain the integer count variable, as you cannot store a `ref` to the integer as a field in the helper class.

You've already seen `Interlocked.Increment` and `Interlocked.Decrement` in action. But what about `Interlocked.Exchange`, which the reporter thread uses? Remember, since multiple threads are attempting to write to the `threadCount` variable, the reporter thread must read the value in a synchronized way as well. That's where `Interlocked.Exchange` comes in. `Interlocked.Exchange`, as its name implies, allows you to exchange the value of a variable with that of another in an atomic fashion, and it returns the value that was stored previously in that location. Since the `Interlocked` class doesn't provide a method to simply read an `Int32` value in an atomic operation, all I'm doing is swapping the `numberThreads` variable's value with its own value, and, as a side effect, the `Interlocked.Exchange` method returns to me the value that was in the slot.

INTERLOCKED METHODS ON SMP SYSTEMS

On Intel symmetric multiprocessing (SMP) platforms and most other SMP systems, simple reads and writes to memory slots that are of the native size are synchronized automatically. On an IA-32 system, reads and writes to properly aligned 32-bit values are synchronized. Therefore, in the previous example where I showed the use of `Interlocked.Exchange` merely to read an `Int32` value, it would not have been necessary if the variable were aligned properly.

By default, the CLR works hard to make sure that values are aligned properly on natural boundaries. However, you can override the placement of values within a class or structure using the `FieldOffsetAttribute` on fields, thus forcing a misaligned data field. If an `Int32` is not aligned, the guarantee mentioned in the previous paragraph is lost. In such a case, you must use `Interlocked.Exchange` to read the value reliably.

The `Interlocked...` methods are all implemented on IA-32 systems using the `lock` prefix. This prefix causes the processor `LOCK#` signal to be asserted. This prevents the other processors in the system from accessing the value concurrently, which is necessary for complex operations where values are incremented and so on. One handy quality of the `lock` prefix is that the misaligned data field does not adversely affect the integrity of the lock. In other words, it works perfectly fine with misaligned data. That's why `Interlocked.Exchange` is the ticket for reading misaligned data atomically.

Finally, consider the fact that the `Interlocked` class implements overloads of some of the methods so that they work with 64-bit values, floating-point numbers, and object references. In fact, `Interlocked...` even offers generic overloads for working with object references. Consider what it means to work with 64-bit values atomically on a 32-bit system. Naturally, there is no possible way to read such values atomically without resorting to the `Interlocked` class. In fact, for this very reason, the .NET 2.0 version of the `Interlocked` class introduced `Interlocked.Read` for `Int64` values. Naturally, such a beast is not necessary on 64-bit systems and should simply boil down to a regular read. However, the CLR is meant to work on multiple platforms, so you should always use `Interlocked.Read` when working with 64-bit values.

For these reasons, it would be better safe than sorry to always use `Interlocked.Exchange` for reading and writing values atomically, since it could prove troublesome to validate that the data is not misaligned and no bigger than the native size prior to reading or writing it in a raw manner. Determining the native size on the platform and basing code conditionally on such data goes against the grain of the cross-platform spirit of managed code.

The last method to cover in the `Interlocked` class is `CompareExchange`. This little method is handy indeed. It's similar to `Interlocked.Exchange`, in that it allows you to exchange the value of a location or slot in an atomic fashion. However, it only does the exchange if the original value compares equal to a provided comparand. In any event, the method always returns the original value. One extremely handy use of the `CompareExchange` method is to create a lightweight *spin lock*. A spin lock gets its name from the fact that if it cannot acquire the lock, it will spin in a tight loop until it can. Typically, when implementing a spin lock, you put your thread to sleep for a very brief slice of time with each failed attempt to acquire the lock. That way, the thread scheduler can give processor

time to another thread while you wait. If you don't want the thread to sleep but only to release its time slice, you can pass a value of 0 to `Thread.Sleep`. Let's look at an example:

```
using System;
using System.IO;
using System.Threading;

public class SpinLock
{
    public SpinLock( int spinWait ) {
        this.spinWait = spinWait;
    }

    public void Enter() {
        while( Interlocked.CompareExchange(ref theLock,
                                           1,
                                           0) == 1 ) {
            // The lock is taken, spin.
            Thread.Sleep( spinWait );
        }
    }

    public void Exit() {
        // Reset the lock.
        Interlocked.Exchange( ref theLock,
                              0 );
    }

    private int theLock = 0;
    private int spinWait;
}

public class SpinLockManager : IDisposable
{
    public SpinLockManager( SpinLock spinLock ) {
        this.spinLock = spinLock;
        spinLock.Enter();
    }

    public void Dispose() {
        spinLock.Exit();
    }

    private SpinLock spinLock;
}

public class EntryPoint
{
    static private Random rnd = new Random();
    private static SpinLock logLock = new SpinLock( 10 );
    private static StreamWriter fsLog =
        new StreamWriter( File.Open("log.txt",
                                    FileMode.Append,
                                    FileAccess.Write,
                                    FileShare.None) );
```

```
    private static void RndThreadFunc() {
        using( new SpinLockManager(logLock) ) {
            fsLog.WriteLine( "Thread Starting" );
            fsLog.Flush();
        }

        int time = rnd.Next( 10, 200 );
        Thread.Sleep( time );

        using( new SpinLockManager(logLock) ) {
            fsLog.WriteLine( "Thread Exiting" );
            fsLog.Flush();
        }
    }

    static void Main() {
        // Start the threads that wait random time.
        Thread[] rndthreads = new Thread[ 50 ];
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
                new Thread( new ThreadStart(
                                EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

This example is similar to the previous one. It creates 50 threads that wait a random amount of time. However, instead of managing a thread count, it outputs a line to a log file. Since this writing is happening from multiple threads, and instance methods of `StreamWriter` are not thread-safe, you must do the writing in a safe manner within the context of a lock. That is where the `SpinLock` class comes in. Internally, it manages a lock variable in the form of an integer, and it uses `Interlocked.CompareExchange` to gate access to the lock. The call to `Interlocked.CompareExchange` in `SpinLock.Enter` is saying

1. If the lock value is equal to 0, replace the value with 1 to indicate that the lock is taken; otherwise, do nothing.
2. If the value of the slot already contains 1, it's taken, and you must sleep and spin.

Both of those items occur in an atomic fashion via the `Interlocked` class, so there is no possible way that more than one thread at a time can acquire the lock. When the `SpinLock.Exit` method is called, all it needs to do is reset the lock. However, that must be done atomically as well—hence, the call to `Interlocked.Exchange`.

In this example, I decided to illustrate the use of the disposable/using idiom to implement deterministic destruction, where you introduce another class—in this case, `SpinLockManager`—to implement the RAII idiom. This saves you from having to remember to write `finally` blocks all over the place. Of course, you still have to remember to use the `using` keyword, but if you follow the idiom more closely than this example, you would implement a finalizer that would assert in the debug build if it ran and the object had not been disposed of.[2]

2. See Chapter 13 for more information on this technique.

Keep in mind that spin locks implemented in this way are not reentrant. Any function that has acquired the lock cannot be called again until it has released the lock. This doesn't mean that you cannot use spin locks with recursive programming techniques. It just means that you must release the lock before recursing, or else suffer a deadlock.

■ **Note** If you require a reentrant wait mechanism, you can use wait objects that are more structured, such as the `Monitor` class, which I cover in the next section, or kernel-based wait objects.

Incidentally, if you'd like to see some fireworks, so to speak, try uncommenting the use of the spin lock in the `RndThreadFunc` method and run the result several times. You'll most likely notice the output in the log file gets a little ugly. The ugliness should increase if you attempt the same test on a multiprocessor machine.

Monitor Class

In the previous section, I showed you how to implement a spin lock using the methods of the `Interlocked` class. A spin lock is not always the most efficient synchronization mechanism, especially if you use it in an environment where a wait is almost guaranteed. The thread scheduler keeps having to wake up the thread and allow it to recheck the lock variable. As I mentioned before, a spin lock is ideal when you need a lightweight, non-reentrant synchronization mechanism and the odds are low that a thread will have to wait in the first place. When you know the likelihood of waiting is high, you should use a synchronization mechanism that allows the scheduler to avoid waking the thread until the lock is available. .NET provides the `System.Threading.Monitor` class to allow synchronization between threads within the same process. You can use this class to guard access to certain variables or to gate access to code that should only be run on one thread at a time.

■ **Note** The Monitor pattern provides a way to ensure synchronization such that only one method, or a block of protected code, executes at one time. A `Mutex` is typically used for the same task. However, `Monitor` is much lighter and faster. `Monitor` is appropriate when you must guard access to code within a single process. `Mutex` is appropriate when you must guard access to a resource from multiple processes.

One potential source of confusion regarding the `Monitor` class is that you cannot instantiate an instance of this class. The `Monitor` class, much like the `Interlocked` class, is merely a containing namespace for a collection of static methods that do the work. If you're used to using critical sections in Win32, you know that at some point you must allocate and initialize a `CRITICAL_SECTION` structure. Then, to enter and exit the lock, you call the Win32 `EnterCriticalSection` and `LeaveCriticalSection` functions. You can achieve exactly the same task using the `Monitor` class in the managed environment. To enter and exit the critical section, you call `Monitor.Enter` and `Monitor.Exit`. Where you pass a `CRITICAL_SECTION` object to the Win32 critical section functions, you pass an object reference to the `Monitor` methods.

Internally, the CLR manages a sync block for every object instance in the process. Basically, it's a flag of sorts, similar to the integer used in the examples of the previous section describing the `Interlocked` class. When you obtain the lock on an object, this flag is set. When the lock is released, this flag is reset. The `Monitor` class is the gateway to accessing this flag. The versatility of this scheme is that every object instance in the CLR potentially contains one of these locks. I say potentially because the CLR allocates them in a lazy fashion, since not every object instance's lock will be uti-

lized. To implement a critical section, all you have to do is create an instance of `Object`. Let's look at an example using the `Monitor` class by modifying the example from the previous section:

```
using System;
using System.Threading;

public class EntryPoint
{
    static private object theLock = new Object();
    static private int numberThreads = 0;
    static private Random rnd = new Random();

    private static void RndThreadFunc() {
        // Manage thread count and wait for a
        // random amount of time between 1 and 12
        // seconds.
        try {
            Monitor.Enter( theLock );
            ++numberThreads;
        }
        finally {
            Monitor.Exit( theLock );
        }

        int time = rnd.Next( 1000, 12000 );
        Thread.Sleep( time );

        try {
            Monitor.Enter( theLock );
            --numberThreads;
        }
        finally {
            Monitor.Exit( theLock );
        }
    }

    private static void RptThreadFunc() {
        while( true ) {
            int threadCount = 0;
            try {
                Monitor.Enter( theLock );
                threadCount = numberThreads;
            }
            finally {
                Monitor.Exit( theLock );
            }

            Console.WriteLine( "{0} thread(s) alive",
                               threadCount );
            Thread.Sleep( 1000 );
        }
    }

    static void Main() {
        // Start the reporting threads.
```

```
        Thread reporter =
            new Thread( new ThreadStart(
                                EntryPoint.RptThreadFunc) );
        reporter.IsBackground = true;
        reporter.Start();

        // Start the threads that wait random time.
        Thread[] rndthreads = new Thread[ 50 ];
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
                new Thread( new ThreadStart(
                                EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

Notice that I perform all access to the `numberThreads` variable within a critical section in the form of an object lock. Before each access, the accessor must obtain the lock on the `theLock` object instance. The type of `theLock` field is of type `object` simply because its actual type is inconsequential. The only thing that matters is that it is a reference type—that is, an instance of `object` rather than a value type. Since you only need the `object` instance to utilize its internal sync block, you can just instantiate an object of type `System.Object`.

One thing you've probably also noticed is that the code is uglier than the version that used the `Interlocked` methods. Whenever you call `Monitor.Enter`, you want to guarantee that the matching `Monitor.Exit` executes no matter what. I mitigated this problem in the examples using the `Interlocked` class by wrapping the usage of the `Interlocked` class methods within a class named `SpinLockManager`. Can you imagine the chaos that could ensue if a `Monitor.Exit` call was skipped because of an exception? Therefore, you always want to utilize a `try/finally` block in these situations. The creators of the C# language recognized that developers were going through a lot of effort to ensure that these `finally` blocks were in place when all they were doing was calling `Monitor.Exit`. So, they made our lives easier by introducing the `lock` keyword. Consider the same example again, this time using the `lock` keyword:

```
using System;
using System.Threading;

public class EntryPoint
{
    static private object theLock = new Object();
    static private int numberThreads = 0;
    static private Random rnd = new Random();

    private static void RndThreadFunc() {
        // Manage thread count and wait for a
        // random amount of time between 1 and 12
        // seconds.
        lock( theLock ) {
            ++numberThreads;
        }

        int time = rnd.Next( 1000, 12000 );
        Thread.Sleep( time );

        lock( theLock ) {
```

```
            --numberThreads;
        }
    }

    private static void RptThreadFunc() {
        while( true ) {
            int threadCount = 0;
            lock( theLock ) {
                threadCount = numberThreads;
            }

            Console.WriteLine( "{0} thread(s) alive",
                               threadCount );
            Thread.Sleep( 1000 );
        }
    }

    static void Main() {
        // Start the reporting threads.
        Thread reporter =
            new Thread( new ThreadStart(
                              EntryPoint.RptThreadFunc) );
        reporter.IsBackground = true;
        reporter.Start();

        // Start the threads that wait random time.
        Thread[] rndthreads = new Thread[ 50 ];
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
                new Thread( new ThreadStart(
                                  EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

Notice that the code is much cleaner now, and in fact, there are no more explicit calls to any `Monitor` methods at all. Under the covers, however, the compiler is expanding the `lock` keyword into the familiar `try/finally` block with calls to `Monitor.Enter` and `Monitor.Exit`. You can verify this by examining the generated IL code using ILDASM.

In many cases, synchronization implemented internally within a class is as simple as implementing a critical section in this manner. But when only one lock object is needed across all methods within the class, you can simplify the model even more by eliminating the extra dummy instance of `System.Object` by using the `this` keyword when acquiring the lock through the `Monitor` class. You'll probably come across this usage pattern often in C# code. Although it saves you from having to instantiate an object of type `System.Object`—which is pretty lightweight, I might add—it does come with its own perils. For example, an external consumer of your object could actually attempt to utilize the sync block within your object by calling `Monitor.Enter` before even calling one of your methods that will try to acquire the same lock. Technically, that's just fine, since the same thread can call `Monitor.Enter` multiple times. In other words, `Monitor` locks are reentrant, unlike the spin locks of the previous section. However, when a lock is released, it must be released by calling `Monitor.Exit` a matching number of times. So, now you have to rely upon the consumers of your object to either use the `lock` keyword or a `try/finally` block to ensure that their call to `Monitor.Enter` is matched appropriately with `Monitor.Exit`. Any time you can avoid such

uncertainty, do so. Therefore, I recommend against locking via the `this` keyword, and I suggest instead using a private instance of `System.Object` as your lock. You could achieve the same effect if there were some way to declare the sync block flag of an object `private`, but alas, that is not possible.

Beware of Boxing

When you're using the `Monitor` methods to implement locking, internally `Monitor` uses the sync block of object instances to manage the lock. Since every object instance can potentially have a sync block, you can use any reference to an object, even an object reference to a boxed value. Even though you can, you should never pass a value type instance to `Monitor.Enter`, as demonstrated in the following code example:

```
using System;
using System.Threading;

public class EntryPoint
{
    static private int counter = 0;

    // NEVER DO THIS !!!
    static private int theLock = 0;

    static private void ThreadFunc() {
        for( int i = 0; i < 50; ++i ) {
            Monitor.Enter( theLock );
            try {
                Console.WriteLine( ++counter );
            }
            finally {
                Monitor.Exit( theLock );
            }
        }
    }

    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        Thread thread2 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        thread1.Start();
        thread2.Start();
    }
}
```

If you attempt to execute this code, you will immediately be presented with a `SynchronizationLockException`, complaining that an object synchronization method was called from an unsynchronized block of code. Why does this happen? In order to find the answer, you need to remember that implicit boxing occurs when you pass a value type to a method that accepts a reference type. And remember, passing the same value type to the same method multiple times will result in a different boxing reference type each time. Therefore, the reference object used within the body of `Monitor.Exit` is different from the one used inside of the body of `Monitor.Enter`. This is another example of how implicit boxing in the C# language can cause you grief. You may have noticed that I used the old `try/finally` approach in this example. That's because the designers of the C# language created the `lock` statement such that it doesn't accept value types. So, if you just

stick to using the lock statement for handling critical sections, you'll never have to worry about inadvertently passing a boxed value type to the Monitor methods.

Pulse and Wait

I cannot overstate the utility of the Monitor methods to implement critical sections. However, the Monitor methods have capabilities beyond that of implementing simple critical sections. You can also use them to implement handshaking between threads, as well as for implementing queued access to a shared resource.

When a thread has entered a locked region successfully, it can give up the lock and enter a waiting queue by calling Monitor.Wait. The first parameter to Monitor.Wait is the object reference whose sync block represents the lock being used. The second parameter is a timeout value. Monitor.Wait returns a Boolean that indicates whether the wait succeeded or if the timeout was reached. If the wait succeeded, the result is true; otherwise, it is false. When a thread that calls Monitor.Wait completes the wait successfully, it leaves the wait state as the owner of the lock again.

If threads can give up the lock and enter into a wait state, there must be some mechanism to tell the Monitor that it can give the lock back to one of the waiting threads as soon as possible. That mechanism is the Monitor.Pulse method. Only the thread that currently holds the lock is allowed to call Monitor.Pulse. When it's called, the thread first in line in the waiting queue is moved to a ready queue. Once the thread that owns the lock releases the lock, either by calling Monitor.Exit or by calling Monitor.Wait, the first thread in the ready queue is allowed to run. The threads in the ready queue include those that are pulsed and those that have been blocked after a call to Monitor.Enter. Additionally, the thread that owns the lock can move all waiting threads into the ready queue by calling Monitor.PulseAll.

There are many fancy synchronization tasks that you can accomplish using the Monitor.Pulse and Monitor.Wait methods. For example, consider the following example that implements a handshaking mechanism between two threads. The goal is to have both threads increment a counter in an alternating manner:

```
using System;
using System.Threading;

public class EntryPoint
{
    static private int counter = 0;

    static private object theLock = new Object();

    static private void ThreadFunc1() {
        lock( theLock ) {
            for( int i = 0; i < 50; ++i ) {
                Monitor.Wait( theLock, Timeout.Infinite );
                Console.WriteLine( "{0} from Thread {1}",
                          ++counter,
                          Thread.CurrentThread.GetHashCode() );
                Monitor.Pulse( theLock );
            }
        }
    }

    static private void ThreadFunc2() {
        lock( theLock ) {
            for( int i = 0; i < 50; ++i ) {
                Monitor.Pulse( theLock );
                Monitor.Wait( theLock, Timeout.Infinite );
```

```
                Console.WriteLine( "{0} from Thread {1}",
                          ++counter,
                          Thread.CurrentThread.GetHashCode() );
            }
        }
    }

    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc1) );
        Thread thread2 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc2) );
        thread1.Start();
        thread2.Start();
    }
}
```

You'll notice that the output from this example shows that the threads increment `counter` in an alternating fashion.

As another example, you could implement a crude thread pool using `Monitor.Wait` and `Monitor.Pulse`. It may be unnecessary to actually do such a thing, since the .NET Framework offers the `ThreadPool` object, which is robust and likely uses optimized I/O completion ports of the underlying OS. For the sake of example, however, I'll show how you could implement a pool of worker threads that wait for work items to be queued:

```
using System;
using System.Threading;
using System.Collections;

public class CrudeThreadPool
{
    static readonly int MAX_WORK_THREADS = 4;
    static readonly int WAIT_TIMEOUT = 2000;

    public delegate void WorkDelegate();

    public CrudeThreadPool() {
        stop = 0;
        workLock = new Object();
        workQueue = new Queue();
        threads = new Thread[ MAX_WORK_THREADS ];

        for( int i = 0; i < MAX_WORK_THREADS; ++i ) {
            threads[i] =
                new Thread( new ThreadStart(this.ThreadFunc) );
            threads[i].Start();
        }
    }

    private void ThreadFunc() {
        lock( workLock ) {
            int shouldStop = 0;
            do {
                shouldStop = Interlocked.Exchange( ref stop,
                                                   stop );
                if( shouldStop == 0 ) {
                    WorkDelegate workItem = null;
```

```
                    if( Monitor.Wait(workLock, WAIT_TIMEOUT) ) {
                        // Process the item on the front of the
                        // queue
                        lock( workQueue ) {
                            workItem =
                                (WorkDelegate) workQueue.Dequeue();
                        }
                        workItem();
                    }
                }
            } while( shouldStop == 0 );
        }
    }

    public void SubmitWorkItem( WorkDelegate item ) {
        lock( workLock ) {
            lock( workQueue ) {
                workQueue.Enqueue( item );
            }

            Monitor.Pulse( workLock );
        }
    }

    public void Shutdown() {
        Interlocked.Exchange( ref stop, 1 );
    }

    private Queue    workQueue;
    private Object   workLock;
    private Thread[] threads;
    private int      stop;
}

public class EntryPoint
{
    static void WorkFunction() {
        Console.WriteLine( "WorkFunction() called on Thread {0}",
                           Thread.CurrentThread.GetHashCode() );
    }

    static void Main() {
        CrudeThreadPool pool = new CrudeThreadPool();
        for( int i = 0; i < 10; ++i ) {
            pool.SubmitWorkItem(
              new CrudeThreadPool.WorkDelegate(
                                  EntryPoint.WorkFunction) );
        }

        pool.Shutdown();
    }
}
```

In this case, the work item is represented by a delegate that neither accepts nor returns any values. When the `CrudeThreadPool` object is created, it creates a pool of threads and starts them running the main work item processing method. That method simply calls `Monitor.Wait` to wait for

an item to be queued. When `SubmitWorkItem` is called, an item is pushed into the queue and it calls `Monitor.Pulse` to release one of the worker threads. Naturally, access to the queue must be synchronized. In this case, the reference type used to sync access is the queue itself. Additionally, the worker threads must not wait forever, because they need to wake up periodically and check a flag to see if they should shut down gracefully. Optionally, you could simply turn the worker threads into background threads by setting the `IsBackground` property inside the `Shutdown` method. However, in that case, the worker threads may be shut down before they're finished processing their work. Depending on your situation, that may or may not be favorable. Notice that I chose to use the `Interlocked` methods to manage the stop flag used to indicate that the worker threads should exit.

■ **Note** Another useful technique for telling threads to shut down is to create a special type of work item that tells a thread to shut down. The trick is that you need to make sure you push as many of these special work items onto the queue as there are threads in the pool.

Locking Objects

The .NET Framework offers several high-level locking objects that you can use to synchronize access to data from multiple threads. I dedicated the previous section entirely to one type of lock: the `Monitor`. However, the `Monitor` class doesn't implement a kernel lock object; rather, it provides access to the sync lock of every .NET object instance. Previously in this chapter, I also covered the primitive `Interlocked` class methods that you can use to implement spin locks. One reason spin locks are considered so primitive is that they are not reentrant and thus don't allow you to acquire the same lock multiple times. Other higher-level locking objects typically do allow that, as long as you match the number of lock operations with release operations. In this section, I want to cover some useful locking objects that the .NET Framework provides.

No matter what type of locking object you use, you should always strive to write code that keeps the lock for the least time possible. For example, if you acquire a lock to access some data within a method that could take quite a bit of time to process that data, acquire the lock only long enough to make a copy of the data on the local stack, and then release the lock as soon as possible. By using this technique, you will ensure that other threads in your system don't block for inordinate amounts of time to access the same data.

ReaderWriterLock

When synchronizing access to shared data between threads, you'll often find yourself in a position where you have several threads reading, or consuming, the data, while only one thread writes, or produces, the data. Obviously, all threads must acquire a lock before they touch the data to prevent the race condition in which one thread writes to the data while another is in the middle of reading it, thus potentially producing garbage for the reader. However, it sure seems inefficient for multiple threads that are merely going to read the data rather than modify it to be locked out from each other. There is no reason why they should not be able to all read the data concurrently without having to worry about stepping on each other's toes.

The `ReaderWriterLock` elegantly avoids this inefficiency. In a nutshell, it allows multiple readers to access the data concurrently, but as soon as one thread needs to write the data, everyone except the writer must get their hands off. `ReaderWriterLock` manages this feat by using two internal queues. One queue is for waiting readers, and the other is for waiting writers. Figure 12-2 shows a high-level block diagram of what the inside of a `ReaderWriterLock` looks like. In this scenario, four threads are running in the system, and currently, none of the threads are attempting to access the data in the lock.

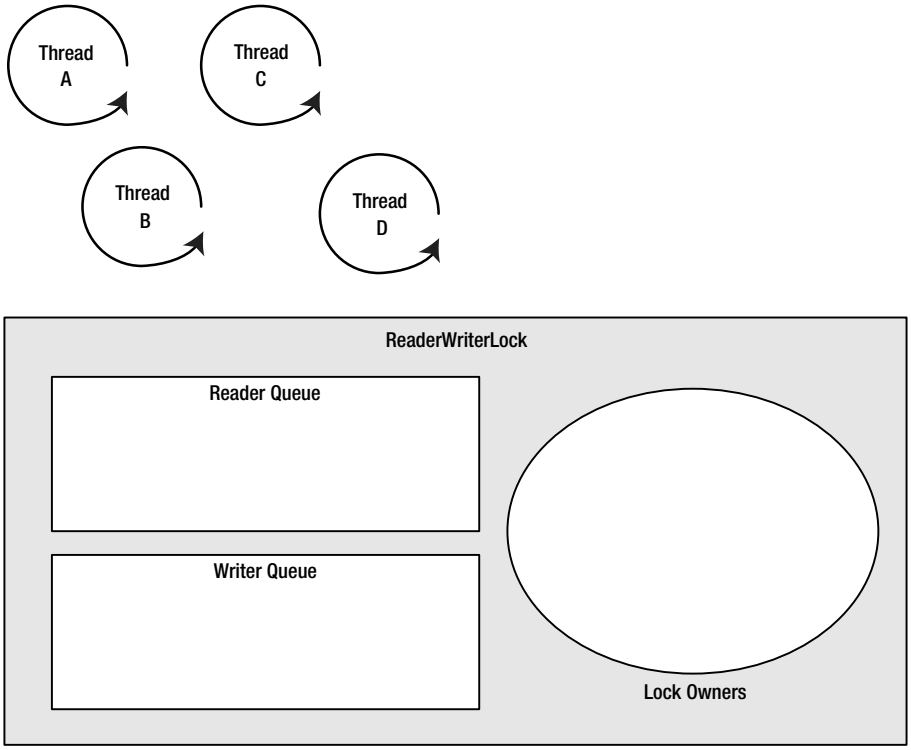

Figure 12-2. *Unutilized ReaderWriterLock*

To access the data, a reader calls `AcquireReaderLock`. Given the state of the lock shown in Figure 12-2, the reader will be placed immediately into the Lock Owners category. Notice the use of plural here, since multiple read lock owners can exist. Things get interesting as soon as one of the threads attempts to acquire the write lock by calling `AcquireWriterLock`. In this case, the writer is placed into the writer queue because readers currently own the lock, as shown in Figure 12-3.

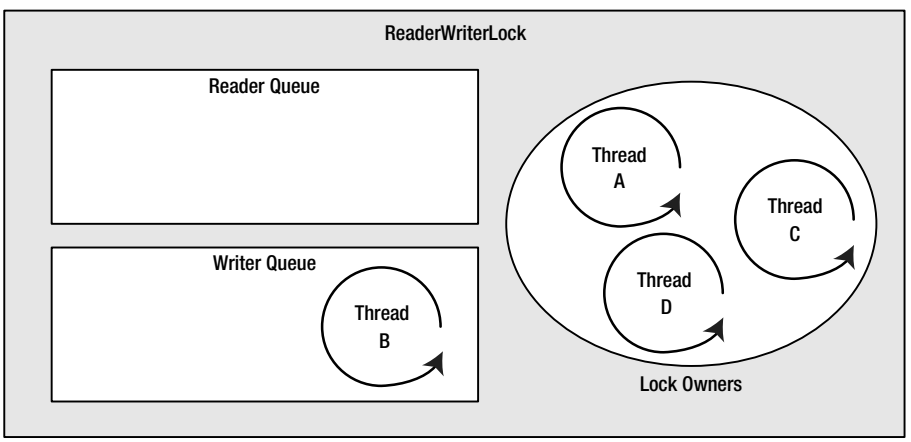

Figure 12-3. *The writer thread is waiting for ReaderWriterLock*

As soon as all of the readers release their lock via a call to `ReleaseReaderLock`, the writer—in this case, Thread B—is allowed to enter the Lock Owners region. But, what happens if Thread A releases its reader lock and then attempts to reacquire the reader lock before the writer has had a chance to acquire the lock? If Thread A were allowed to reacquire the lock, then any thread waiting in the writer queue could potentially be starved of any time with the lock. In order to avoid this, any thread that attempts to require the read lock while a writer is in the queue is placed into the reader queue, as shown in Figure 12-4.

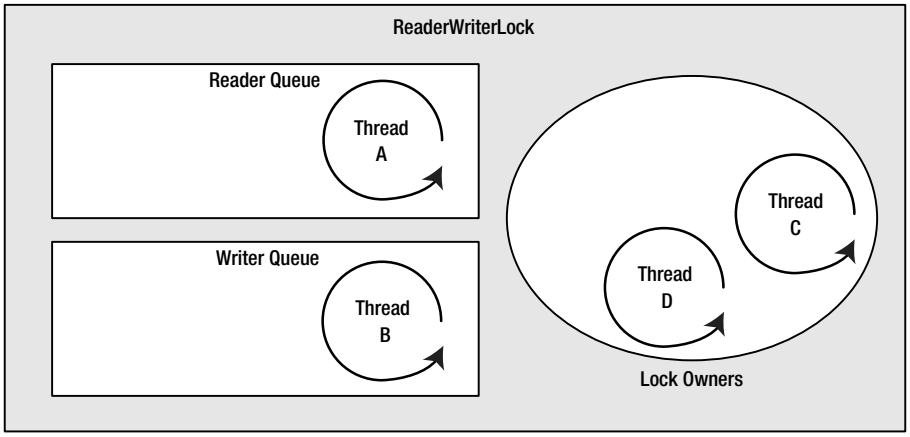

Figure 12-4. *Reader attempting to reacquire lock*

Naturally, this scheme gives preference to the writer queue. That makes sense given the fact that you'd want any readers to get the most up-to-date information. Of course, had the thread that needs the writer lock called `AcquireWriterLock` while the `ReaderWriterLock` was in the state shown in Figure 12-2, it would have been placed immediately into the Lock Owners category without having to go through the writer queue.

The `ReaderWriterLock` is reentrant. Therefore, a thread can call any one of the lock-acquisition methods multiple times, as long as it calls the matching release method the same number of times. Each time the lock is reacquired, an internal lock count is incremented. It should seem obvious that a single thread cannot own both the reader and the writer lock at the same time, nor can it wait in both queues in the `ReaderWriterLock`. It is possible, however, for a thread to upgrade or downgrade the type of lock it owns. For example, if a thread currently owns a reader lock and calls `UpgradeToWriterLock`, its reader lock is released no matter what the lock count is, and then it is placed into the writer queue. The `UpgradeToWriterLock` returns an object of type `LockCookie`. You should hold on to this object and pass it to `DowngradeFromWriterLock` when you're done with the write operation. The `ReaderWriterLock` uses the cookie to restore the reader lock count on the object. Even though you can increase the writer lock count once you've acquired it via `UpgradeToWriterLock`, your call to `DowngradeFromWriterLock` will release the writer lock no matter what the write lock count is. Therefore, it's best that you avoid relying on the writer lock count within an upgraded writer lock.

As with just about every other synchronization object in the .NET Framework, you can provide a timeout with almost every lock acquisition method. This timeout is given in milliseconds. However, instead of the methods returning a `Boolean` to indicate whether the lock was acquired successfully, these methods throw an exception of type `ApplicationException` if the timeout expires. So, if you pass in any timeout value other than `Timeout.Infinite` to one of these functions, be sure to make the call inside a `try` block to catch the potential exception.

ReaderWriterLockSlim

.NET 3.5 introduces a new style of reader/writer lock called `ReaderWriterLockSlim`. It brings a few enhancements to the table, including better deadlock protection, better efficiency, and disposability. It also does not support recursion by default, which adds to its efficiency. If you need recursion, `ReaderWriterLockSlim` provides an overloaded constructor that accepts a value from the `LockRecursionPolicy` enumeration. Microsoft recommends using `ReaderWriterLockSlim` rather than `ReaderWriterLock` for any new development.

With respect to `ReaderWriterLockSlim`, there are four states that the thread can be in:

- Unheld
- Read mode
- Upgradeable mode
- Write mode

Unheld means that the thread is not attempting to read or write to the resource at all. If a thread is in read mode, it has read access to the resource after successfully calling the `EnterReadLock` method. Likewise, if a thread is in write mode, it has write access to the thread after successfully calling `EnterWriteLock`. Just as with `ReaderWriterLock`, only one thread can be in write mode at a time and while any thread is in write mode, all threads are blocked from entering read mode. Naturally, a thread attempting to enter write mode is blocked while any threads still remain in read mode. Once they all exit, the thread waiting for write mode is released. So what is upgradeable mode?

Upgradeable mode is useful if you have a thread that needs read access to the resource but may also from time to time need write access to the resource. Without upgradeable mode, the thread would need to exit read mode and then attempt to enter write mode sequentially. During the time when it is in the unheld mode, another thread could enter read mode, thus stalling the thread attempting to gain the write lock. Only one thread at a time may be in upgradeable mode, and it enters upgradeable mode via a call to `EnterUpgradeableReadLock`. Upgradeable threads may enter read mode or write mode recursively, even for `ReaderWriterLockSlim` instances that were created with recursion turned off. In essence, upgradeable mode is a more powerful form of read mode that allows greater efficiency when entering write mode. If a thread attempts to enter upgradeable mode and another thread is in write mode or threads are in a queue to enter write mode, the thread calling `EnterUpgradeableReadLock` will block until the other thread has exited write mode and the queued threads have entered and exited write mode. This is identical behavior to threads attempting to enter read mode.

`ReaderWriterLockSlim` may throw a `LockRecursionException` in certain circumstances. Since `ReaderWriterLockSlim` instances don't support recursion by default, attempting to call `EnterReadLock`, `EnterWriteLock`, or `EnterUpgradeableReadLock` multiple times from the same thread will result in one of these exceptions. Additionally, whether the instance supports recursion or not, a thread that is already in upgradeable mode and attempts to call `EnterReadLock` or a thread that is in write mode and attempts to call `EnterReadLock` could deadlock the system, so a `LockRecursionException` is thrown in those cases too.

If you're familiar with the `Monitor` class, you may recognize the idiom represented in the method names of `ReaderWriterLockSlim`. Each time a thread enters a state, it must call one of the `Enter...`methods, and each time it leaves that state, it must call one of the corresponding `Exit...` methods. Additionally, just like `Monitor`, `ReaderWriterLockSlim` provides methods that allow you to try to enter the lock without potentially blocking forever with methods such as `TryEnterReadLock`, `TryEnterUpgradeableReadLock`, and `TryEnterWriteLock`. Each of the `Try...` methods allows you to pass in a timeout value indicating how long you are willing to wait.

The general guideline when using `Monitor` is not to use `Monitor` directly, but rather indirectly through the C# `lock` keyword. That's so that you don't have to worry about forgetting to call `Monitor.Exit` and you don't have to type out a `finally` block to ensure that `Monitor.Exit` is called under all circumstances. Unfortunately, there is no equivalent mechanism available to make it easier to enter and exit locks using `ReaderWriterLockSlim`. Always be careful to call the `Exit...` method when you are finished with a lock, and call it from within a `finally` block so that it gets called even in the face of exceptional conditions.

Mutex

The `Mutex` object is a heavier type of lock that you can use to implement mutually exclusive access to a resource. The .NET Framework supports two types of `Mutex` implementations. If it's created without a name, you get what's called a local mutex. But if you create it with a name, the `Mutex` is usable across multiple processes and implemented using a Win32 kernel object, which is one of the heaviest types of lock objects. By that, I mean that it is the slowest and carries the most overhead when used to guard a protected resource from multiple threads. Other lock types, such as the `ReaderWriterLock` and the `Monitor` class, are strictly for use within the confines of a single process. Therefore, for efficiency, you should only use a `Mutex` object when you really need to synchronize execution or access to some resource across multiple processes.

As with other high-level synchronization objects, the `Mutex` is reentrant. When your thread needs to acquire the exclusive lock, you call the `WaitOne` method. As usual, you can pass in a timeout value expressed in milliseconds when waiting for the `Mutex` object. The method returns a `Boolean` that will be `true` if the wait is successful, or `false` if the timeout expired. A thread can call the `WaitOne` method as many times as it wants, as long as it matches the calls with the same amount of `ReleaseMutex` calls.

Since you can use `Mutex` objects across multiple processes, each process needs a way to identify the `Mutex`. Therefore, you can supply an optional name when you create a `Mutex` instance. Providing a name is the easiest way for another process to identify and open the mutex. Since all `Mutex` names exist in the global namespace of the entire operating system, it is important to give the mutex a sufficiently unique name so that it won't collide with `Mutex` names created by other applications. I recommend using a name that is based on the string form of a GUID generated by `GUIDGEN.exe`.

■ **Note** I mentioned that the names of kernel objects are global to the entire machine. That statement is not entirely true if you consider Windows fast user switching and Terminal Services. In those cases, the namespace that contains the name of these kernel objects is instanced for each logged-in user. For times when you really do want your name to exist in the global namespace, you can prefix the name with the special string "`\Global`". For more information, reference *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000* by Mark E. Russinovich and David A. Solomon (Redmond, WA: Microsoft Press, 2004).

If everything about the `Mutex` object sounds strikingly familiar to those of you who are native Win32 developers, that's because the underlying mechanism is, in fact, the Win32 `Mutex` object. In fact, you can get your hands on the actual OS handle via the `SafeWaitHandle` property inherited from the `WaitHandle` base class. I have more to say about the `WaitHandle` class in the "Win32 Synchronization Objects and WaitHandle" section, where I discuss its pros and cons. It's important to note that since you implement the `Mutex` using a kernel mutex, you incur a transition to kernel mode any time you manipulate or wait upon the `Mutex`. Such transitions are extremely slow and should be minimized if you're running time-critical code.

■ **Tip** Avoid using kernel mode objects for synchronization between threads in the same process if at all possible. Prefer more lightweight mechanisms, such as the `Monitor` class or the `Interlocked` class. When effectively synchronizing threads between multiple processes, you have no choice but to use kernel objects. On my current test machine, a simple test showed that using the `Mutex` took more than 44 times longer than the `Interlocked` class and 34 times longer than the `Monitor` class.

Semaphore

The .NET Framework supports semaphores via the `System.Threading.Semaphore` class. They are used to allow a countable number of threads to acquire a resource simultaneously. Each time a thread enters the semaphore via `WaitOne` or any of the other `Wait...` methods, the semaphore count is decremented. When an owning thread calls `Release`, the count is incremented. If a thread attempts to enter the semaphore when the count is zero, it will block until another thread calls `Release`. When a thread calls `Release`, the count is incremented.

Just as with `Mutex`, when you create a semaphore, you may or may not provide a name by which other processes may identify it. If you create it without a name, you end up with a local semaphore that is only useful within the same process. Either way, the underlying implementation uses a Win32 semaphore kernel object. Therefore, it is a very heavy synchronization object that is slow and inefficient. You should prefer local semaphores over named semaphore unless you need to synchronize access across multiple processes for security reasons.

Note that a thread can enter a semaphore recursively. However, it must call `Release` the appropriate number of times to restore the availability count on the semaphore. The task of matching the `Wait...` method calls and subsequent calls to `Release` is entirely up to you. There is nothing in place to keep you from calling `Release` too many times. If you do, then when another thread later calls `Release`, it could attempt to push the count above the allowable limit, at which point it will throw a `SemaphoreFullException`. These bugs are very difficult to find because the point of failure is disjoint from the point of error.

Events

In the .NET Framework, you can use two types to signal events: `ManualResetEvent` and `AutoResetEvent`. As with the `Mutex` object, these event objects map directly to Win32 event objects. If you're familiar with using Win32 events, you'll feel right at home with the .NET event objects. Similar to `Mutex` objects, working with event objects incurs a slow transition to kernel mode. Both event types become signaled when someone calls the `Set` method on an event instance. At that point, a thread waiting on the event will be released. Threads wait for an event by calling the inherited `WaitHandle.WaitOne` method, which is the same method you call to wait on a `Mutex` to become signaled.

I was careful in stating that a waiting thread is released when the event becomes signaled. It's possible that multiple threads could be released when an event becomes signaled. That, in fact, is the difference between `ManualResetEvent` and `AutoResetEvent`. When a `ManualResetEvent` becomes signaled, all threads waiting on it are released. It stays signaled until someone calls its `Reset` method. If any thread calls `WaitOne` while the `ManualResetEvent` is already signaled, then the wait is immediately completed successfully. On the other hand, `AutoResetEvent` objects only release one waiting thread and then immediately reset to the unsignaled set automatically. You can imagine that all threads waiting on the `AutoResetEvent` are waiting in a queue, where only the first thread in the queue is released when the event becomes signaled. However, even though it's useful to assume

that the waiting threads are in a queue, you cannot make any assumptions about which waiting thread will be released first. `AutoResetEvents` are also known as *sync events* based on this behavior.

Using the `AutoResetEvent` type, you could implement a crude thread pool where several threads wait on an `AutoResetEvent` signal to be told that some piece of work is available. When a new piece of work is added to the work queue, the event is signaled to turn one of the waiting threads loose. Implementing a thread pool this way is not efficient and comes with its problems. For example, things become tricky to handle when all threads are busy and work items are pushed into the queue, especially if only one thread is allowed to complete one work item before going back to the waiting queue. If all threads are busy and, say, five work items are queued in the meantime, the event will be signaled but no threads will be waiting. The first thread back into the waiting queue will be released once it calls `WaitOne`, but the others will not, even though four more work items exist in the queue. One solution to this problem is not to allow work items to be queued while all of the threads are busy. That's not really a solution, because it defers some of the synchronization logic to the thread attempting to queue the work item by forcing it to do something appropriate in reaction to a failed attempt to queue a work item. In reality, creating an efficient thread pool is tricky business, to say the least. Therefore, I recommend you utilize the `ThreadPool` class before attempting such a feat. I cover the `ThreadPool` class in detail in the "Using ThreadPool" section.

Since .NET event objects are based on Win32 event objects, you can use them to synchronize execution between multiple processes. Along with the `Mutex`, they are also more inefficient than an alternative, such as the `Monitor` class, because of the kernel mode transition involved. However, the creators of `ManualResetEvent` and `AutoResetEvent` did not expose the ability to name the event objects in their constructors, as they do for the `Mutex` object. Therefore, if you need to create a named event, you should use the `EventWaitHandle` class instead.

Win32 Synchronization Objects and WaitHandle

In the previous two sections, I covered the `Mutex`, `ManualResetEvent`, and `AutoResetEvent` objects. Each one of these types is derived from `WaitHandle`, a general mechanism that you can use in the .NET Framework to manage any type of Win32 synchronization object that you can wait upon. That includes more than just events and mutexes. No matter how you obtain the Win32 object handle, you can use a `WaitHandle` object to manage it. I prefer to use the word *manage* rather than *encapsulate*, because the `WaitHandle` class doesn't do a great job of encapsulation, nor was it meant to. It's simply meant as a wrapper to help you avoid a lot of direct calls to Win32 via the P/Invoke layer when dealing with OS handles.

■ **Note** Take some time to understand when and how to use `WaitHandle`, because many APIs have yet to be mapped into the .NET Framework, and many of them may never be.

I've already discussed the `WaitOne` method used to wait for an object to become signaled. However, the `WaitHandle` class has two handy static methods that you can use to wait on multiple objects. The first is `WaitHandle.WaitAny`. You pass it an array of `WaitHandle` objects, and when any one of the objects becomes signaled, the `WaitAny` method returns an integer indexing into the array to the object that became signaled. The other method is `WaitHandle.WaitAll`, which, as you can imagine, won't return until all of the objects becomes signaled. Both of these methods have defined overloads that accept a timeout value. In the case of a call to `WaitAny` that times out, the return value will be equal to the `WaitHandle.WaitTimeout` constant. In the case of a call to `WaitAll`, a `Boolean` is returned, which is either `true` to indicate that all of the objects became signaled, or `false` to indicate that the wait timed out.

Prior to the existence of the EventWaitHandle class, in order to get a named event, you had to create the underlying Win2 object and then wrap it with a WaitHandle, as I've done in the following example:

```
using System;
using System.Threading;
using System.Runtime.InteropServices;
using System.ComponentModel;
using Microsoft.Win32.SafeHandles;

public class NamedEventCreator
{
    [DllImport( "KERNEL32.DLL", EntryPoint="CreateEventW",
                SetLastError=true )]
    private static extern SafeWaitHandle CreateEvent(
                                    IntPtr lpEventAttributes,
                                    bool   bManualReset,
                                    bool   bInitialState,
                                    string lpName );

    public const int INVALID_HANDLE_VALUE = -1;

    public static AutoResetEvent CreateAutoResetEvent(
                                        bool initialState,
                                        string name ) {
        // Create named event.
        SafeWaitHandle rawEvent = CreateEvent( IntPtr.Zero,
                                       false,
                                       false,
                                       name );
        if( rawEvent.IsInvalid ) {
            throw new Win32Exception(
                            Marshal.GetLastWin32Error() );
        }

        // Create a managed event type based on this handle.
        AutoResetEvent autoEvent = new AutoResetEvent( false );

        // Must clean up handle currently in autoEvent
        // before swapping it with the named one.
        autoEvent.SafeWaitHandle = rawEvent;

        return autoEvent;
    }
}
```

Here I've used the P/Invoke layer to call down into the Win32 CreateEventW function to create a named event. Several things are worth noting in this example. For instance, I've completely punted on the handle security, just as the rest of the .NET Framework standard library classes tend to do. Therefore, the first parameter to CreateEvent is IntPtr.Zero, which is the best way to pass a NULL pointer to the Win32 error. Notice that you detect the success or failure of the event creation by testing the IsInvalid property on the SafeWaitHandle. When you detect this value, you throw a Win32Exception type. You then create a new AutoResetEvent to wrap the raw handle just created. WaitHandle exposes a property named SafeWaitHandle, whereby you can modify the underlying Win32 handle of any WaitHandle derived type.

■ **Note** You may have noticed the legacy `Handle` property in the documentation. You should avoid this property, since reassigning it with a new kernel handle won't close the previous handle, thus resulting in a resource leak unless you close it yourself. You should use `SafeHandle` derived types instead. The `SafeHandle` type also uses constrained execution regions to guard against resource leaks in the event of an asynchronous exception such as `ThreadAbortException`. You can read more about constrained execution regions in Chapter 7.

In the previous example, you can see that I declared the `CreateEvent` method to return a `SafeWaitHandle`. Although it's not obvious from the documentation of `SafeWaitHandle`, it has a private default constructor that the P/Invoke layer is capable of using to create and initialize an instance of this class.

Be sure to check out the rest of the `SafeHandle` derived types in the `Microsoft.Win32.SafeHandles` namespace. Specifically, the .NET 2.0 Framework introduced `SafeHandleMinusOneIsInvalid` and `SafeHandleZeroOrMinusOneIsInvalid` for convenience when defining your own Win32-based `SafeWaitHandle` derivatives.

Be aware that the `WaitHandle` type implements the `IDisposable` interface. Therefore, you want to make judicious use of the `using` keyword in your code whenever using `WaitHandle` instances or instances of any of the classes that derive from it, such as `Mutex`, `AutoResetEvent`, and `ManualResetEvent`.

One last thing that you need to be aware of when using `WaitHandle` objects and those objects that derive from the type is that you cannot abort or interrupt managed threads in a timely manner when they're blocked via a method to `WaitHandle`. Since the actual OS thread that is running under the managed thread is blocked inside the OS—thus outside of the managed execution environment—it can only be aborted or interrupted as soon as it reenters the managed environment. Therefore, if you call `Abort` or `Interrupt` on one of those threads, the operation will be pended until the thread completes the wait at the OS level. You want to be cognizant of this when you block using a `WaitHandle` object in managed threads.

Using ThreadPool

A thread pool is ideal in a system where small units of work are performed regularly in an asynchronous manner. A good example is a web server or any other kind of server listening for requests on a port. When a request comes in, a new thread is given the request and processes it. The server achieves a high level of concurrency and optimal utilization by servicing these requests in multiple threads. Typically, the slowest operation on a computer is an I/O operation. Storage devices, such as hard drives, are very slow in comparison to the processor and its ability to access memory. Therefore, to make optimal use of the system, you want to begin other work items while it's waiting on an I/O operation to complete in another thread. Creating a thread pool to manage such a system is an amazing task fraught with many details and pitfalls. However, the .NET environment exposes a prebuilt, ready-to-use thread pool via the `ThreadPool` class.

The `ThreadPool` class is similar to the `Monitor` and `Interlocked` classes in the sense that you cannot actually create instances of the `ThreadPool` class. Instead, you use the static methods of the `ThreadPool` class to manage the thread pool that each process gets by default in the CLR. In fact, you don't even have to worry about creating the thread pool. It gets created when it is first used. If you have used thread pools in the Win32 world, whether it be via the system thread pool that was introduced in Windows 2000 or via I/O completion ports, you'll notice that the .NET thread pool is the same beast with a managed interface placed on top of it.

To queue an item to the thread pool, you simply call `ThreadPool.QueueUserWorkItem`, passing it an instance of the `WaitCallback` delegate. The thread pool gets created the first time your process

calls this function. The callback method that is called through the `WaitCallback` delegate accepts a reference to `System.Object` and has a return type of `void`. The object reference is an optional context object that the caller can supply to an overload of `QueueUserWorkItem`. If you don't provide a context, the context reference will be `null`. Once the work item is queued, a thread in the thread pool will execute the callback as soon as it becomes available. Once a work item is queued, it cannot be removed from the queue except by a thread that will complete the work item. So if you need to cancel a work item, you must craft a way to let your callback know that it should do nothing once it gets called.

The thread pool is tuned to keep the machine processing work items in the most efficient way possible. It uses an algorithm based upon how many CPUs are available in the system to determine how many threads to create in the pool. However, even once it computes how many threads to create, the thread pool may, at times, contain more threads than originally calculated. For example, suppose the algorithm decides that the thread pool should contain four threads. Then, suppose the server receives four requests that access a backend database that takes some time. If a fifth request comes in during this time, no threads will be available to dispatch the work item. What's worse, the four busy threads are just sitting around waiting for the I/O to complete. In order to keep the system running at peak performance, the thread pool will actually create another thread when it knows all of the others are blocking. After the work items have all been completed and the system is in a steady state again, the thread pool will then kill off any extra threads created like this. Even though you cannot easily control how many threads are in a thread pool, you can easily control the minimum number of threads that are idle in the pool waiting for work via calls to `GetMinThreads` and `SetMinThreads`.

I urge you to read the details of the `System.Threading.ThreadPool` static methods in the MSDN documentation if you plan to deal directly with the thread pool. In reality, it's rare that you'll ever need to insert work items directly into the thread pool. There is another, more elegant, entry point into the thread pool via delegates and asynchronous procedure calls, which I cover in the next section.

Asynchronous Method Calls

Although you can manage the work items put into the thread pool directly via the `ThreadPool` class, a more popular way to employ the thread pool is via asynchronous delegate calls. When you declare a delegate, the CLR defines a class for you that derives from `System.MulticastDelegate`. One of the methods defined is the `Invoke` method, which takes exactly the same function signature as the delegate definition. The C# language, of course, offers a syntactical shortcut to calling the `Invoke` method. In fact, you cannot explicitly call the `Invoke` method in C#. But along with `Invoke`, the CLR also defines two methods, `BeginInvoke` and `EndInvoke`, that are at the heart of the asynchronous processing pattern used throughout the CLR. This pattern is similar to the IOU pattern introduced earlier in the chapter.

The basic idea is probably evident from the names of the methods. When you call the `BeginInvoke` method on the delegate, the operation is pended to be completed in another thread. When you call the `EndInvoke` method, the results of the operation are given back to you. If the operation has not completed at the time you call `EndInvoke`, the calling thread blocks until the operation is complete. Let's look at a short example that shows the general pattern in use. Suppose you have a method that computes your taxes for the year, and you want to call it asynchronously because it could take a reasonably long amount of time to do:

```
using System;
using System.Threading;

public class EntryPoint
{
    // Declare the delegate for the async call.
    private delegate Decimal ComputeTaxesDelegate( int year );
```

```
    // The method that computes the taxes.
    private static Decimal ComputeTaxes( int year ) {
        Console.WriteLine( "Computing taxes in thread {0}",
                           Thread.CurrentThread.GetHashCode() );

        // Here's where the long calculation happens.
        Thread.Sleep( 6000 );

        // You owe the man.
        return 4356.98M;
    }

    static void Main() {
        // Let's make the asynchronous call by creating
        // the delegate and calling it.
        ComputeTaxesDelegate work =
            new ComputeTaxesDelegate( EntryPoint.ComputeTaxes );
        IAsyncResult pendingOp = work.BeginInvoke( 2004,
                                                   null,
                                                   null );

        // Do some other useful work.
        Thread.Sleep( 3000 );

        // Finish the async call.
        Console.WriteLine( "Waiting for operation to complete." );
        Decimal result = work.EndInvoke( pendingOp );

        Console.WriteLine( "Taxes owed: {0}", result );
    }
}
```

The first thing you will notice with the pattern is that the `BeginInvoke` method's signature does not match that of the `Invoke` method. That's because you need some way to identify the particular work item that you just pended with the call to `BeginInvoke`. Therefore, `BeginInvoke` returns a reference to an object that implements the `IAsyncResult` interface. This object is like a cookie that you can hold on to so that you can identify the work item in progress. Through the methods on the `IAsyncResult` interface, you can check on the status of the operation, such as whether it is completed. I'll discuss this interface in more detail in a bit, along with the extra two parameters added onto the end of the `BeginInvoke` method declaration for which I'm passing `null`. When the thread that requested the operation is finally ready for the result, it calls `EndInvoke` on the delegate. However, since the method must have a way to identify which asynchronous operation to get the results for, you must pass in the object that you got back from the `BeginInvoke` method. In this example, you'll notice the call to `EndInvoke` blocking for some time as the operation completes.

■ **Note** If an exception is generated while the delegate's target code is running asynchronously in the thread pool, the exception is rethrown when the initiating thread makes a call to `EndInvoke`.

Part of the beauty of the IOU asynchronous pattern that delegates implement is that the called code doesn't even need to be aware of the fact that it's getting called asynchronously. Of course, it's rarely practical to call a method asynchronously when it was never designed to be, if it touches data in the system that other methods touch without using any synchronization mechanisms. Nonetheless, the headache of creating an asynchronous calling infrastructure around the method has been

mitigated by the delegate generated by the CLR, along with the per-process thread pool. Moreover, the initiator of the asynchronous action doesn't even need to be aware of how the asynchronous behavior is implemented.

Now let's look a little closer at the `IAsyncResult` interface for the object returned from the `BeginInvoke` method. The interface declaration looks like the following:

```
public interface IAsyncResult
{
  Object AsyncState { get; }
  WaitHandle AsyncWaitHandle { get; }
  bool CompletedSynchronously { get; }
  bool IsCompleted { get; }
}
```

In the previous example, I chose to wait for the computation to finish by calling `EndInvoke`. I could have instead waited on the `WaitHandle` returned by the `IAsyncResult.AsyncWaitHandle` property before calling `EndInvoke`. The end result would have been the same in this case. However, the fact that the `IAsyncResult` interface exposes the `WaitHandle` allows you to have multiple threads in the system wait for this one action to complete if they needed to.

Two other properties allow you to query whether the operation has completed. The `IsCompleted` property simply returns a `Boolean` representing the fact. You could construct a polling loop that checks this flag repeatedly. However, that would be much more inefficient than just waiting on the `WaitHandle`. Nonetheless, it is there if you need it. Another `Boolean` property is `CompletedSynchronously`. The asynchronous processing pattern in the .NET Framework provides for the option that the call to `BeginInvoke` could actually choose to process the work synchronously rather than asynchronously. The `CompletedSynchronously` property allows you to determine if this happened. As it is currently implemented, the CLR will never do such a thing when delegates are called asynchronously, but this could change at any time. However, since it is recommended that you apply this same asynchronous pattern whenever you design a type that can be called asynchronously, the capability was built into the pattern. For example, suppose you have a class where a method to process generalized operations synchronously is supported. If one of those operations simply returns the version number of the class, then you know that operation can be done quickly, and you may choose to perform it synchronously.

Finally, the `AsyncState` property of `IAsyncResult` allows you to attach any type of specific context data to an asynchronous call. This is the last of the extra two parameters added at the end of the `BeginInvoke` signature. In my previous example, I passed in `null` because I didn't need to use it. Although I chose to harvest the result of the operation via a call to `EndInvoke`, I could have chosen to be notified via a callback. Consider the following modifications to the previous example:

```
using System;
using System.Threading;

public class EntryPoint
{
    // Declare the delegate for the async call.
    private delegate Decimal ComputeTaxesDelegate( int year );

    // The method that computes the taxes.
    private static Decimal ComputeTaxes( int year ) {
        Console.WriteLine( "Computing taxes in thread {0}",
                           Thread.CurrentThread.GetHashCode() );

        // Here's where the long calculation happens.
        Thread.Sleep( 6000 );
```

```
        // You owe the man.
        return 4356.98M;
    }

    private static void TaxesComputed( IAsyncResult ar ) {
        // Let's get the results now.
        ComputeTaxesDelegate work =
            (ComputeTaxesDelegate) ar.AsyncState;

        Decimal result = work.EndInvoke( ar );
        Console.WriteLine( "Taxes owed: {0}", result );
    }

    static void Main() {
        // Let's make the asynchronous call by creating
        // the delegate and calling it.
        ComputeTaxesDelegate work =
            new ComputeTaxesDelegate( EntryPoint.ComputeTaxes );
        work.BeginInvoke( 2004,
                          new AsyncCallback(
                                EntryPoint.TaxesComputed),
                          work );

        // Do some other useful work.
        Thread.Sleep( 3000 );

        Console.WriteLine( "Waiting for operation to complete." );
        Thread.Sleep( 4000 );
    }
}
```

Now, instead of calling `EndInvoke` from the thread that called `BeginInvoke`, I'm requesting that the thread pool call the `TaxesComputed` method via an instance of the `AsyncCallback` delegate that I passed in as the second-to-last parameter of `BeingInvoke`. Using a callback to process the result completes the asynchronous processing pattern by allowing the thread that started the operation to continue to work without ever having to explicitly wait on the worker thread. Notice that the `TaxesComputed` callback method must still call `EndInvoke` to harvest the results of the asynchronous call. In order to do that, though, it must have an instance of the delegate. That's where the `IAsyncResult.AsyncState` context object comes in handy. In my example, I initialize it to point to the delegate by passing the delegate as the last parameter to `BeginInvoke`. The main thread that calls `BeginInvoke` has no need for the object returned by the call, since it never actively polls the state of the operation, nor does it wait explicitly for the operation to complete. The added sleep at the end of the `Main` method is there for the sake of the example. Remember, all threads in the thread pool run as background threads. Therefore, if you don't wait at this point, the process would exit long before the operation completes. If you need asynchronous work to occur in a foreground thread, it is best to create a new class that implements the asynchronous pattern of `BeingInvoke/EndInvoke` and use a foreground thread to do the work. Never change the background status of a thread in the thread pool via the `IsBackground` property on the current thread. Even if you try, you'll find that it has no effect.

■ **Note** It's important to realize that when your asynchronous code is executing and when the callback is executing, you are running in an arbitrary thread context. You cannot make any assumptions about which thread is running your code. In many respects, this technique is similar to driver development on Windows platforms.

Using a callback to handle the completion of a work item is very handy when creating a server process that will handle incoming requests. For example, suppose you have a process that listens on a specific TCP/IP port for an incoming request. When it receives one, it replies with the requested information. To achieve high utilization, you definitely want to pend these operations asynchronously. Consider the following example that listens on port 1234 and when it receives anything at all, it simply replies with "Hello World!":

```
using System;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;

public class EntryPoint {
    private const int CONNECT_QUEUE_LENGTH = 4;
    private const int LISTEN_PORT = 1234;

    static void ListenForRequests() {
        Socket listenSock =
            new Socket( AddressFamily.InterNetwork,
                        SocketType.Stream,
                        ProtocolType.Tcp );
        listenSock.Bind( new IPEndPoint(IPAddress.Any,
                                        LISTEN_PORT) );
        listenSock.Listen( CONNECT_QUEUE_LENGTH );

        while( true ) {
            using( Socket newConnection = listenSock.Accept() ) {
                // Send the data.
                byte[] msg =
                    Encoding.UTF8.GetBytes( "Hello World!" );
                newConnection.Send( msg, SocketFlags.None );
            }
        }
    }

    static void Main() {
        // Start the listening thread.
        Thread listener = new Thread(
                            new ThreadStart(
                                EntryPoint.ListenForRequests) );
        listener.IsBackground = true;
        listener.Start();

        Console.WriteLine( "Press <enter> to quit" );
        Console.ReadLine();
    }
}
```

This example creates an extra thread that simply loops around listening for incoming connections and servicing them as soon as they come in. The problems with this approach are many. First, only one thread handles the incoming connections. If the connections are flying in at a rapid rate, it will quickly become overwhelmed. Think about a web server that could easily see thousands of requests per second. As it turns out, the `Socket` class implements the asynchronous calling pattern of the .NET Framework. Using the pattern, you can make the server a little bit better by servicing the incoming requests using the thread pool, as follows:

```
using System;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;

public class EntryPoint {
    private const int CONNECT_QUEUE_LENGTH = 4;
    private const int LISTEN_PORT = 1234;

    static void ListenForRequests() {
        Socket listenSock =
            new Socket( AddressFamily.InterNetwork,
                        SocketType.Stream,
                        ProtocolType.Tcp );
        listenSock.Bind( new IPEndPoint(IPAddress.Any,
                                        LISTEN_PORT) );
        listenSock.Listen( CONNECT_QUEUE_LENGTH );

        while( true ) {
            Socket newConnection = listenSock.Accept();
            byte[] msg = Encoding.UTF8.GetBytes( "Hello World!" );
            newConnection.BeginSend( msg,
                                     0, msg.Length,
                                     SocketFlags.None,
                                     null, null );
        }
    }

    static void Main() {
        // Start the listening thread.
        Thread listener = new Thread(
                             new ThreadStart(
                                 EntryPoint.ListenForRequests) );
        listener.IsBackground = true;
        listener.Start();

        Console.WriteLine( "Press <enter> to quit" );
        Console.ReadLine();
    }
}
```

The server is becoming a little more efficient, since it is now sending the data to the incoming connection from a thread in the thread pool. This code also demonstrates a fire-and-forget strategy when using the asynchronous pattern. The caller is not interested in the return object that implements `IAsyncResult`, nor is it interested in setting a callback method to get called when the work completes. This fire-and-forget call is a valiant attempt to make the server more efficient. However, the result is less than satisfactory, since the `using` statement from the previous incarnation of the server is gone. The `Socket` is not closed in a timely manner, and the remote connections are held open until the GC gets around to finalizing the `Socket` objects. Therefore, the asynchronous call needs to include a callback in order to close the connection. It wouldn't make sense for the listening thread to wait on the `EndSend` method, as that would put you back in the same inefficiency boat you were in before.

■ **Note** When you get an object that implements `IAsyncResult` back from starting an asynchronous operation, that object must implement the `IAsyncResult.AsyncWaitHandle` property to allow users to obtain a handle they can wait on. In the case of `Socket`, an instance of `OverlappedAsyncResult` is returned. That class ultimately derives from `System.Net.LazyAsyncResult`. It doesn't actually create the event to wait on until someone accesses it via the `IAsyncResult.AsyncWaitHandle` property. This lazy creation spares the burden of creating a lock object that goes unused most of the time. Also, it is the responsibility of the `OverlappedAsyncResult` object to close the OS handle when it is finished with it.

However, before getting to the callback, consider the listening thread for a moment. All it does is spin around listening for incoming requests. Wouldn't it be more efficient if the server were to use the thread pool to handle the listening too? Of course it would! So, now, let me present the new and improved "Hello World!" server that makes full use of the process thread pool:

```
using System;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;

public class EntryPoint {
    private const int CONNECT_QUEUE_LENGTH = 4;
    private const int LISTEN_PORT = 1234;
    private const int MAX_CONNECTION_HANDLERS = 4;

    private static void HandleConnection( IAsyncResult ar ) {
        Socket listener = (Socket) ar.AsyncState;

        Socket newConnection = listener.EndAccept( ar );
        byte[] msg = Encoding.UTF8.GetBytes( "Hello World!" );
        newConnection.BeginSend( msg,
                                 0, msg.Length,
                                 SocketFlags.None,
                                 new AsyncCallback(
                                      EntryPoint.CloseConnection),
                                 newConnection );

        // Now queue another accept.
        listener.BeginAccept(
            new AsyncCallback(EntryPoint.HandleConnection),
            listener );
    }

    static void CloseConnection( IAsyncResult ar ) {
        Socket theSocket = (Socket) ar.AsyncState;
        theSocket.Close();
    }

    static void Main() {
        Socket listenSock =
            new Socket( AddressFamily.InterNetwork,
                        SocketType.Stream,
                        ProtocolType.Tcp );
        listenSock.Bind( new IPEndPoint(IPAddress.Any,
                                        LISTEN_PORT) );
```

```
        listenSock.Listen( CONNECT_QUEUE_LENGTH );

        // Pend the connection handlers.
        for( int i = 0; i < MAX_CONNECTION_HANDLERS; ++i ) {
            listenSock.BeginAccept(
                new AsyncCallback(EntryPoint.HandleConnection),
                listenSock );
        }

        Console.WriteLine( "Press <enter> to quit" );
        Console.ReadLine();
    }
}
```

Now, the "Hello World" server is making full use of the process thread pool and can handle incoming client requests with the best concurrency. Incidentally, testing the connection is fairly simple using the built-in Windows Telnet client. Simply run Telnet from a command prompt or from the Start ➤ Run dialog, and at the prompt enter the following command to connect to port 1234 on the local machine while the server process is running in another command window:

```
Microsoft Telnet> open 127.0.0.1 1234
```

Timers

Yet another entry point into the thread pool is via `Timer` objects in the `System.Threading` namespace. As the name implies, you can arrange for the thread pool to call a delegate at a specific time as well as at regular intervals. Let's look at an example of how to use a `Timer` object:

```
using System;
using System.Threading;

public class EntryPoint
{
    private static void TimerProc( object state ) {
        Console.WriteLine( "The current time is {0} on thread {1}",
                           DateTime.Now,
                           Thread.CurrentThread.GetHashCode() );
        Thread.Sleep( 3000 );
    }

    static void Main() {
        Console.WriteLine( "Press <enter> when finished\n\n" );

        Timer myTimer =
            new Timer( new TimerCallback(EntryPoint.TimerProc),
                       null,
                       0,
                       2000 );

        Console.ReadLine();
        myTimer.Dispose();
    }
}
```

When the timer is created, you must give it a delegate to call at the required time. Therefore, I've created a `TimerCallback` delegate that points back to the static `TimerProc` method. The second

parameter to the `Timer` constructor is an arbitrary state object that you can pass in. When your timer callback is called, this state object is passed to the timer callback. In my example, I have no need for a state object, so I just pass `null`. The last two parameters to the constructor define when the callback gets called. The second-to-last parameter indicates when the timer should fire for the first time. In my example, I pass 0, which indicates that it should fire immediately. The last parameter is the period at which the callback should be called. In my example, I've asked for a two-second period. If you don't want the timer to be called periodically, pass `Timeout.Infinite` as the last parameter. Finally, to shut down the timer, simply call its `Dispose` method.

In my example, you may wonder why I have the `Sleep` call inside the `TimerProc` method. It's there just to illustrate a point, and that is that an arbitrary thread calls the `TimerProc`. Therefore, any code that executes as a result of your `TimerCallback` delegate must be thread-safe. In my example, the first thread in the thread pool to call `TimerProc` sleeps longer than the next timeout, so the thread pool calls the `TimerProc` method two seconds later on another thread, as you can see in the generated output. You could really cause some strain on the thread pool if you were to notch up the sleep in the `TimerProc`.

■ **Note** If you've ever used the `Timer` class in the `System.Windows.Forms` namespace, you must realize that it's a completely different beast than the `Timer` class in the `System.Threading` namespace. For one, the `Forms.Timer` is based upon Win32 Windows messaging—namely, the `WM_TIMER` message. One handy quality of the `Forms.Timer` is that its timer callback is always called on the same thread. However, the only way that happens in the first place is if the UI thread that the timer is a part of has an underlying UI message pump. If the pump stalls, so do the `Forms.Timer` callbacks. So, naturally, the `Threading.Timer` is more powerful in the sense that it doesn't suffer from this dependency. However, the drawback is that you must code your `Threading.Timer` callbacks in a thread-safe manner.

Summary

In this chapter, I've covered the intricacies of managed threads in the .NET environment. I covered the various mechanisms in place for managing synchronization between threads, including the `Interlocked`, `Monitor`, `AutoResetEvent`, `ManualResetEvent`, `WaitHandle`-based objects, and so on. I then described the IOU pattern and how the .NET Framework uses it extensively to get work done asynchronously. That discussion centered around the CLR's usage of the `ThreadPool` based upon the Windows thread pool implementation.

Threading always adds complexity to applications. However, when used properly, it can make applications more responsive to user commands and more efficient. Although multithreading development comes with its pitfalls, the .NET Framework and the CLR mitigate many of those risks and provide a model that shields you from the intricacies of the operating system—most of the time. For example, thread pools have always been difficult to implement, even after a common implementation was added to the Windows operating system. Not only does the .NET environment provide a nice buffer between your code and the Windows thread pool intricacies, but it also allows your code to run on other platforms that implement the .NET Framework, such as the Mono runtime running on Linux. If you understand the details of the threading facilities provided by the .NET runtime and are familiar with multithreaded synchronization techniques, as covered in this chapter, then you're well on your way to producing effective multithreaded applications.

In the next chapter, I go in search of a C# canonical form for types. I investigate the checklist of questions you should ask yourself when designing any type using C# for the .NET Framework.