CHAPTER 28

I/O-Bound Asynchronous Operations

In this chapter:

The previous chapter focused on ways to perform compute-bound operations asynchronously, allowing the thread pool to schedule the tasks onto multiple cores so that multiple threads can work concurrently, which increases throughput while using system resources efficiently. In this chapter, we'll focus on performing I/O-bound operations asynchronously, allowing hardware devices to handle the tasks so that threads and the CPU are not used at all. However, the thread pool still plays an important role because, as you'll see, the thread pool threads will process the results of the various I/O operations.

How Windows Performs I/O Operations

Let's begin by discussing how Windows performs synchronous I/O operations. Figure 28-1 represents a computer system with several hardware devices connected to it. Each of these hardware devices has its own circuit board, each of which contains a small, special-purpose computer that knows how to control its hardware device. For example, the hard disk drive has a circuit board that knows how to spin up the drive, seek the head to the right track, read or write data from or to the disk, and transfer the data to or from your computer's memory.

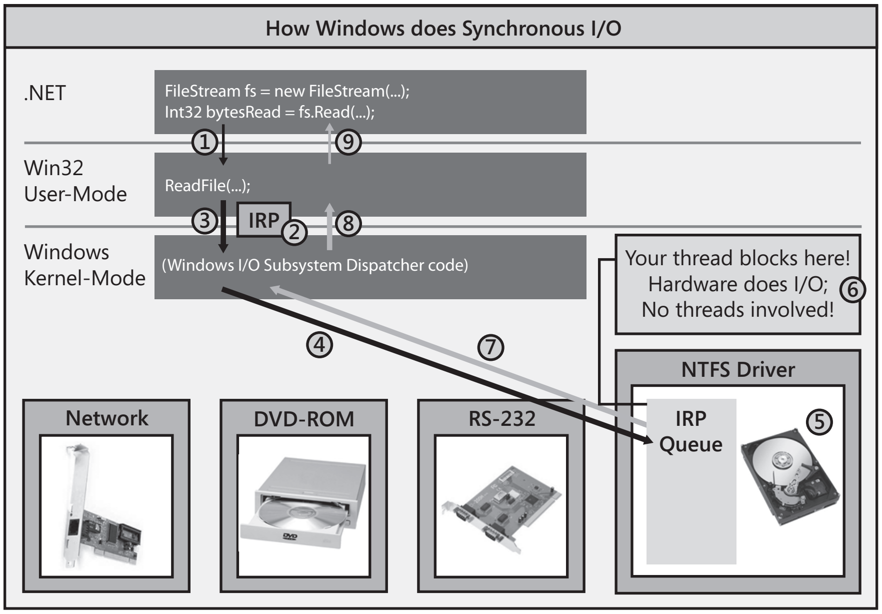
How Windows does Synchronous I/O
.NET
FileStream fs = new FileStream(...);
Int32 bytesRead = fs.Read(...);
Win32 User-Mode
ReadFile(...);
IRP
Windows Kernel-Mode
(Windows I/O Subsystem Dispatcher code)
Your thread blocks here! Hardware does I/O; No threads involved!
NTFS Driver
IRP Queue
Network
DVD-ROM
RS-232
1
2
3
4
5
6
7
8
9

FIGURE 28-1 How Windows performs a synchronous I/O operation.

In your program, you open a disk file by constructing a `FileStream` object. Then you call the `Read` method to read data from the file. When you call `FileStream`'s `Read` method, your thread transitions from managed code to native/user-mode code and `Read` internally calls the Win32 `ReadFile` function (#1). `ReadFile` then allocates a small data structure called an I/O Request Packet (IRP) (#2). The IRP structure is initialized to contain the handle to the file, an offset within the file where bytes will start to be read from, the address of a `Byte[]` that should be filled with the bytes being read, the number of bytes to transfer, and some other less interesting stuff.

`ReadFile` then calls into the Windows kernel by having your thread transition from native/user-mode code to native/kernel-mode code, passing the IRP data structure to the kernel (#3). From the device handle in the IRP, the Windows kernel knows which hardware device the I/O operation is destined for, and Windows delivers the IRP to the appropriate device driver's IRP queue (#4). Each device driver maintains its own IRP queue that contains I/O requests from all processes running on the machine. As IRP packets show up, the device driver passes the IRP information to the circuit board associated with the actual hardware device. The hardware device now performs the requested I/O operation (#5).

But here is the important part: While the hardware device is performing the I/O operation, your thread that issued the I/O request has nothing to do, so Windows puts your thread to sleep so that it is not wasting CPU time (#6). This is great, but although your thread is not wasting time, it is wasting space (memory), as its user-mode stack, kernel-mode stack, thread environment block (TEB), and other data structures are sitting in memory but are not being accessed at all. In addition, for GUI applications, the UI can't respond to user input while the thread is blocked. All of this is bad.

Ultimately, the hardware device will complete the I/O operation, and then Windows will wake up your thread, schedule it to a CPU, and let it return from kernel mode to user mode, and then back to managed code (#7, #8, and #9). `FileStream`'s `Read` method now returns an `Int32`, indicating the actual number of bytes read from the file so that you know how many bytes you can examine in the `Byte[]` that you passed to `Read`.

Let's imagine that you are implementing a web application and as each client request comes in to your server, you need to make a database request. When a client request comes in, a thread pool thread will call into your code. If you now issue a database request synchronously, the thread will block for an indefinite amount of time waiting for the database to respond with the result. If during this time another client request comes in, the thread pool will have to create another thread and again this thread will block when it makes another database request. As more and more client requests come in, more and more threads are created, and all these threads block waiting for the database to respond. The result is that your web server is allocating lots of system resources (threads and their memory) that are barely even used!

And to make matters worse, when the database does reply with the various results, threads become unblocked and they all start executing. But because you might have lots of threads running and relatively few CPU cores, Windows has to perform frequent context switches, which hurts performance even more. This is no way to implement a scalable application.

Now, let's discuss how Windows performs asynchronous I/O operations. In Figure 28-2, I have removed all the hardware devices except the hard disk from the picture, I introduce the common language runtime's (CLR's) thread pool, and I've modified the code slightly. I still open the disk file by constructing a `FileStream` object, but now I pass in the `FileOptions.Asynchronous` flag. This flag tells Windows that I want my read and write operations against the file to be performed asynchronously.

To read data from the file, I now call `ReadAsync` instead of `Read`. `ReadAsync` internally allocates a `Task<Int32>` object to represent the pending completion of the read operation. Then, `ReadAsync` calls Win32's `ReadFile` function (#1). `ReadFile` allocates its IRP, initializes it just like it did in the synchronous scenario (#2), and then passes it down to the Windows kernel (#3). Windows adds the IRP to the hard disk driver's IRP queue (#4), but now, instead of blocking your thread, your thread is allowed to return to your code; your thread immediately returns from its call to `ReadAsync` (#5, #6, and #7). Now, of course, the IRP has not necessarily been processed yet, so you cannot have code after `ReadAsync` that attempts to access the bytes in the passed-in `Byte[]`.

Now you might ask, when and how do you process the data that will ultimately be read? Well, when you call `ReadAsync`, it returns to you a `Task<Int32>` object. Using this object, you can call `ContinueWith` to register a callback method that should execute when the task completes and then process the data in this callback method. Or, alternatively, you can use C#'s asynchronous function feature to simplify your code by allowing you to write it sequentially (as you would if you were performing synchronous I/O).

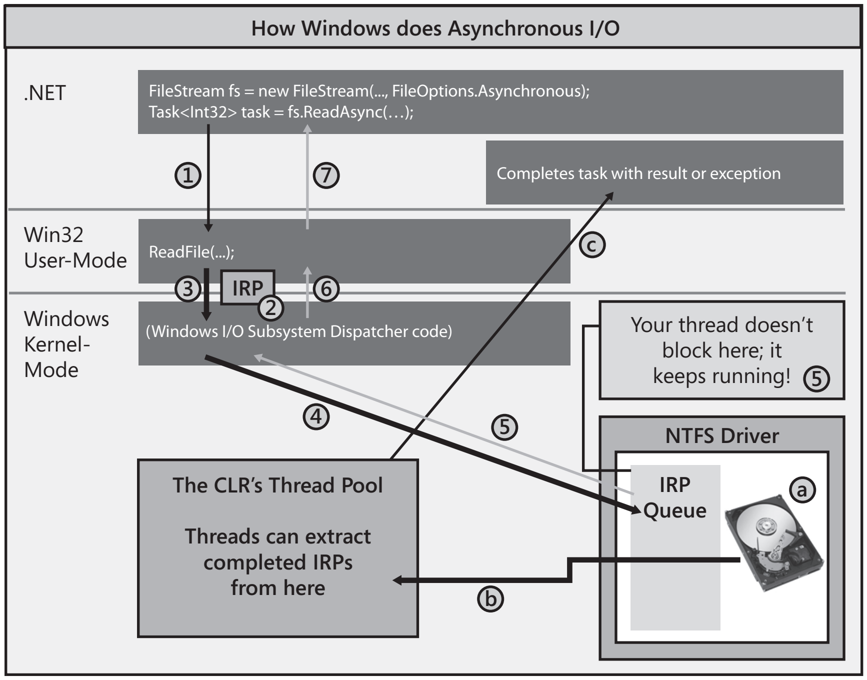
How Windows does Asynchronous I/O
.NET
FileStream fs = new FileStream(..., FileOptions.Asynchronous);
Task<Int32> task = fs.ReadAsync(…);
①
⑦
Completes task with result or exception
Win32 User-Mode
ReadFile(...);
ⓒ
③
IRP
②
⑥
Windows Kernel-Mode
(Windows I/O Subsystem Dispatcher code)
Your thread doesn't block here; it keeps running! ⑤
④
⑤
NTFS Driver
IRP Queue
ⓐ
The CLR's Thread Pool
Threads can extract completed IRPs from here
ⓑ

FIGURE 28-2 How Windows performs an asynchronous I/O operation.

When the hardware device completes processing the IRP (a), it will queue the completed IRP into the CLR's thread pool (b). Sometime in the future, a thread pool thread will extract the completed IRP and execute code that completes the task by setting an exception (if an error occurred) or the result (in this case, an `Int32` indicating the number of bytes successfully read) (c).[1] So now the `Task` object knows when the operation has completed and this, in turn, lets your code run so it can safely access the data inside the `Byte[]`.

Now that you understand the basics, let's put it all into perspective. Let's say that a client request comes in, and our server makes an asynchronous database request. As a result, our thread won't block, and it will be allowed to return to the thread pool so that it can handle more incoming client requests. So now we have just one thread handling all incoming client requests. When the database server responds, its response is also queued into the thread pool, so our thread pool thread will just process it at some point and ultimately send the necessary data back to the client. At this point, we have just one thread processing all client requests and all database responses. Our server is using very few system resources and it is still running as fast as it can, especially because there are no context switches!

If items appear in the thread pool quicker than our one thread can process them all, then the thread pool might create additional threads. The thread pool will quickly create one thread per CPU

[1] Completed IRPs are extracted from the thread pool by using a first-in-first-out (FIFO) algorithm.

on the machine. So, on a quad-processor machine, four client requests/database responses (in any combination) are running on four threads without any context switching.[2]

However, if any of these threads voluntarily block (by invoking a synchronous I/O operation, calling `Thread.Sleep`, or waiting to acquire a thread synchronization lock), then Windows notifies the thread pool that one of its threads has stopped running. The thread pool now realizes that the CPUs are undersaturated and creates a new thread to replace the blocked thread. This, of course, is not ideal because creating a new thread is very expensive in terms of both time and memory.

What's worse is that the blocked thread might wake up and now the CPUs are oversaturated again and context switching must occur, decreasing performance. However, the thread pool is smart here. As threads complete their processing and return to the pool, the thread pool won't let them process new work items until the CPUs become exactly saturated again, thereby reducing context switches and improving performance. And if the thread pool later determines that it has more threads in it than it needs, it lets the extra threads kill themselves, thereby reclaiming the resources that these threads were using.

Internally, the CLR's thread pool uses a Windows resource called an *I/O Completion Port* to elicit the behavior that I've just described. The CLR creates an I/O Completion Port when it initializes and, as you open hardware devices, these devices can be bound to the I/O Completion Port so that device drivers know where to queue the completed IRPs. If you want to understand more about this mechanism, I recommend the book, *Windows via C/C*++, Fifth Edition, by myself and Christophe Nasarre (Microsoft Press, 2007).

In addition to minimal resource usage and reduced context switches, we get many other benefits when performing I/O operations asynchronously. Whenever a garbage collection starts, the CLR must suspend all the threads in the process. Therefore, the fewer threads we have, the faster the garbage collector runs. In addition, when a garbage collection occurs, the CLR must walk all the threads' stacks looking for roots. Again, the fewer threads there are, the fewer stacks there are, and this also makes the garbage collection faster. But, in addition, if our threads don't block while processing work items, the threads tend to spend most of their time waiting in the thread pool. So when a garbage collection occurs, the threads are at the top of their stack, and walking each thread's stack for roots takes very little time.

Also, when you debug an application, Windows suspends all threads in the debuggee when you hit a breakpoint. Then, when you continue executing the debuggee, Windows has to resume all its threads, so if you have a lot of threads in an application, single-stepping through the code in a debugger can be excruciatingly slow. Using asynchronous I/O allows you to have just a few threads, improving your debugging performance.

[2] This is assuming that other threads are not running on the computer, which is true most of the time, because most computers are running at far less than 100-percent CPU usage. And, even if CPU usage is at 100 percent due to threads lower than priority 8, your application will not have its responsiveness and performance impacted because your application's thread will just pre-empt the lower priority threads. If other threads are running whose priority interferes with your thread's priorities, then context switching does occur. This is bad for performance reasons, but it is good for overall application responsiveness reasons. Remember that Windows gives each process at least one thread and performs context switches to ensure that an application whose thread is an infinite loop doesn't stop other applications' threads from running.

And, here's yet another benefit: let's say that your application wants to download 10 images from various websites, and that it takes 5 seconds to download each image. If you perform this work synchronously (downloading one image after another), then it takes you 50 seconds to get the 10 images. However, if you use just one thread to initiate 10 asynchronous download operations, then all 10 are being performed concurrently and all 10 images will come back in just 5 seconds! That is, when performing multiple synchronous I/O operations, the time it takes to get all the results is the sum of the times required for each individual result. However, when performing multiple asynchronous I/O operations, the time it takes to get all the results is the time required to get the single worst-performing operation.

For GUI applications, asynchronous operations offer yet another advantage: the application's user interface doesn't hang and remains responsive to the end user. In fact, if you are building a Microsoft Silverlight or Windows Store application, you must perform all I/O operations asynchronously, because the class libraries available to you for performing I/O operations only expose these operations asynchronously; the equivalent synchronous methods simply do not exist in the library. This was done purposely ensuring that these applications can never issue a synchronous I/O operation, thereby blocking the GUI thread making the application nonresponsive to the end user. This forces developers to build responsive applications providing end users a better experience.

C#'s Asynchronous Functions

Performing asynchronous operations is the key to building scalable and responsive applications that allow you to use very few threads to execute lots of operations. And when coupled with the thread pool, asynchronous operations allow you to take advantage of all of the CPUs that are in the machine. Realizing the enormous potential here, Microsoft designed a programming model that would make it easy for developers to take advantage of this capability.[3] This pattern leverages `Tasks` (as discussed in Chapter 27, "Compute-Bound Asynchronous Operations") and a C# language feature called asynchronous functions (or async functions, for short). Here is an example of code that uses an async function to issue two asynchronous I/O operations.

```
private static async Task<String> IssueClientRequestAsync(String serverName, String message) {
   using (var pipe = new NamedPipeClientStream(serverName, "PipeName", PipeDirection.InOut,
      PipeOptions.Asynchronous | PipeOptions.WriteThrough)) {

      pipe.Connect(); // Must Connect before setting ReadMode
      pipe.ReadMode = PipeTransmissionMode.Message;

      // Asynchronously send data to the server
      Byte[] request = Encoding.UTF8.GetBytes(message);
      await pipe.WriteAsync(request, 0, request.Length);
```

[3] For developers using a version of the Microsoft .NET Framework prior to version 4.5, my `AsyncEnumerator` class (that is part of my Power Threading library available on *http://Wintellect.com/*) allows you to use a programming model quite similar to the programming model that now ships as part of .NET Framework 4.5. In fact, the success of my `AsyncEnumerator` class allowed me to assist Microsoft in designing the programming model I explain in this chapter. Due to the similarities, it is trivial to migrate code using my `AsyncEnumerator` class to the new programming model.

```
      // Asynchronously read the server's response
      Byte[] response = new Byte[1000];
      Int32 bytesRead = await pipe.ReadAsync(response, 0, response.Length);
      return Encoding.UTF8.GetString(response, 0, bytesRead);
   }  // Close the pipe
}
```

In the preceding code, you can tell that IssueClientRequestAsync is an async function, because I specified async on the first line just after static. When you mark a method as async, the compiler basically transforms your method's code into a type that implements a state machine (the details of which will be discussed in the next section). This allows a thread to execute some code in the state machine and then return without having the method execute all the way to completion. So, when a thread calls IssueClientRequestAsync, the thread constructs a NamedPipeClientStream, calls Connect, sets its ReadMode property, converts the passed-in message to a Byte[] and then calls WriteAsync. WriteAsync internally allocates a Task object and returns it back to IssueClientRequestAsync. At this point, the C# await operator effectively calls ContinueWith on the Task object passing in the method that resumes the state machine and then, the thread returns from IssueClientRequestAsync.

Sometime in the future, the network device driver will complete writing the data to the pipe and then, a thread pool thread will notify the Task object, which will then activate the ContinueWith callback method, causing a thread to resume the state machine. More specifically, a thread will re-enter the IssueClientRequestAsync method but at the point of the await operator. Our method now executes compiler-generated code that queries the status of the Task object. If the operation failed, an exception representing the failure is thrown. If the operation completes successfully, the await operator returns the result. In this case, WriteAsync returns a Task instead of a Task<TResult>, so there is no return value.

Now, our method continues executing by allocating a Byte[] and then calls NamedPipeClientStream's asynchronous ReadAsync method. Internally, ReadAsync creates a Task<Int32> object and returns it. Again, the await operator effectively calls ContinueWith on the Task<Int32> object passing in the method that resumes the state machine. And then, the thread returns from IssueClientRequestAsync again.

Sometime in the future, the server will send a response back to the client machine, the network device driver gets this response, and a thread pool thread notifies the Task<Int32> object, which will then resume the state machine. The await operator causes the compiler to generate code that queries the Task object's Result property (an Int32) and assigns the result to the bytesRead local variable or throws an exception if the operation failed. Then, the rest of the code in IssueClientRequestAsync executes, returning the result string and closing the pipe. At this point, the state machine has run to completion and the garbage collector will reclaim any memory it needed.

Because async functions return before their state machine has executed all the way to completion, the method calling IssueClientRequestAsync will continue its execution right after IssueClientRequestAsync executes its first await operator. But, how can the caller know when IssueClientRequestAsync has completed executing its state machine in its entirety? Well, when you mark a method as async, the compiler automatically generates code that creates a Task object when the state machine begins its execution; this Task object is completed automatically when the state machine runs to completion. You'll notice that the IssueClientRequestAsync method's return type is a Task<String>. It actually returns the Task<String> object that the compiler-generated code creates back to its caller, and the Task's Result property is of type String in this case. Near the bottom of IssueClientRequestAsync, I return a string. This causes the compiler-generated code to complete the Task<String> object it created and set its Result property to the returned string.

You should be aware of the following restrictions related to async functions:

- You cannot turn your application's Main method into an async function. In addition, constructors, property accessor methods and event accessor methods cannot be turned into async functions.
- You cannot have any out or ref parameters on an async function.
- You cannot use the await operator inside a catch, finally, or unsafe block.
- You cannot take a lock that supports thread ownership or recursion before an await operator and release it after the await operator. The reason is because one thread might execute the code before the await and a different thread might execute the code after the await. If you use await within a C# lock statement, the compiler issues an error. If you explicitly call Monitor's Enter and Exit methods instead, then the code will compile but Monitor.Exit will throw a SynchronizationLockException at run time.[4]
- Within a query expression, the await operator may only be used within the first collection expression of the initial from clause or within the collection expression of a join clause.

These restrictions are pretty minor. If you violate one, the compiler will let you know, and you can usually work around the problem with some small code modifications.

How the Compiler Transforms an Async Function into a State Machine

When working with async functions, you will be more productive with them if you have an understanding and appreciation for the code transform that the compiler is doing for you. And, I think the easiest and best way for you to learn that is by going through an example. So, let's start off by defining some simple type definitions and some simple methods.

```
internal sealed class Type1 { }
internal sealed class Type2 { }
private static async Task<Type1> Method1Async() {
   /* Does some async thing that returns a Type1 object */
}
private static async Task<Type2> Method2Async() {
   /* Does some async thing that returns a Type2 object */
}
```

[4] Instead of blocking a thread by having it wait on a thread synchronization construct, you could await the task returned from calling SemaphoreSlim's WaitAsync method or my own OneManyLock's AcquireAsync method. I discuss both of these in Chapter 30, "Hybrid Thread Synchronization Constructs."

Now, let me show you an async function that consumes these simple types and methods.

```
private static async Task<String> MyMethodAsync(Int32 argument) {
   Int32 local = argument;
   try {
      Type1 result1 = await Method1Async();
      for (Int32 x = 0; x < 3; x++) {
         Type2 result2 = await Method2Async();
      }
   }
   catch (Exception) {
      Console.WriteLine("Catch");
   }
   finally {
      Console.WriteLine("Finally");
   }
   return "Done";
}
```

Although `MyMethodAsync` seems rather contrived, it demonstrates some key things. First, it is an async function itself that returns a `Task<String>` but the code's body ultimately returns a `String`. Second, it calls other functions that execute operations asynchronously, one stand-alone and the other from within a `for` loop. Finally, it also contains exception handling code. When compiling `My-MethodAsync`, the compiler transforms the code in this method to a state machine structure that is capable of being suspended and resumed.

I took the preceding code, compiled it, and then reverse engineered the IL code back into C# source code. I then simplified the code and added a lot of comments to it so you can understand what the compiler is doing to make async functions work. The following is the essence of the code created by the compiler's transformation. I show the transformed `MyMethodAsync` method as well as the state machine structure it now depends on.

```
// AsyncStateMachine attribute indicates an async method (good for tools using reflection);
// the type indicates which structure implements the state machine
[DebuggerStepThrough, AsyncStateMachine(typeof(StateMachine))]
private static Task<String> MyMethodAsync(Int32 argument) {
   // Create state machine instance & initialize it
   StateMachine stateMachine = new StateMachine() {
      // Create builder returning Task<String> from this stub method
      // State machine accesses builder to set Task completion/exception
      m_builder = AsyncTaskMethodBuilder<String>.Create(),

      m_state = -1,           // Initialize state machine location
      m_argument = argument   // Copy arguments to state machine fields
   };

   // Start executing the state machine
   stateMachine.m_builder.Start(ref stateMachine);
   return stateMachine.m_builder.Task; // Return state machine's Task
}

// This is the state machine structure
[CompilerGenerated, StructLayout(LayoutKind.Auto)]
private struct StateMachine : IAsyncStateMachine {
   // Fields for state machine's builder (Task) & its location
   public AsyncTaskMethodBuilder<String> m_builder;
   public Int32 m_state;

   // Argument and local variables are fields now:
   public Int32 m_argument, m_local, m_x;
   public Type1 m_resultType1;
   public Type2 m_resultType2;

   // There is 1 field per awaiter type.
   // Only 1 of these fields is important at any time. That field refers
   // to the most recently executed await that is completing asynchronously:
   private TaskAwaiter<Type1> m_awaiterType1;
   private TaskAwaiter<Type2> m_awaiterType2;

   // This is the state machine method itself
   void IAsyncStateMachine.MoveNext() {
      String result = null;   // Task's result value

      // Compiler-inserted try block ensures the state machine’s task completes
      try {
         Boolean executeFinally = true;   // Assume we're logically leaving the 'try' block
         if (m_state == -1) {             // If 1st time in state machine method,
            m_local = m_argument;         // execute start of original method
         }

         // Try block that we had in our original code
         try {
            TaskAwaiter<Type1> awaiterType1;
            TaskAwaiter<Type2> awaiterType2;

            switch (m_state) {
               case -1: // Start execution of code in 'try'
                  // Call Method1Async and get its awaiter
                  awaiterType1 = Method1Async().GetAwaiter();
                  if (!awaiterType1.IsCompleted) {
                     m_state = 0;                   // 'Method1Async' is completing
                                                    // asynchronously
                     m_awaiterType1 = awaiterType1; // Save the awaiter for when we come back

                     // Tell awaiter to call MoveNext when operation completes
                     m_builder.AwaitUnsafeOnCompleted(ref awaiterType1, ref this);
                     // The line above invokes awaiterType1's OnCompleted which approximately
                     // calls ContinueWith(t => MoveNext()) on the Task being awaited.
                     // When the Task completes, the ContinueWith task calls MoveNext

                     executeFinally = false;        // We're not logically leaving the 'try'
                                                    // block
                     return;                        // Thread returns to caller
                  }
                  // 'Method1Async' completed synchronously
                  break;

               case 0:  // 'Method1Async' completed asynchronously
                  awaiterType1 = m_awaiterType1;  // Restore most-recent awaiter
                  break;
               case 1:  // 'Method2Async' completed asynchronously
                  awaiterType2 = m_awaiterType2;  // Restore most-recent awaiter
                  goto ForLoopEpilog;
            }

            // After the first await, we capture the result & start the 'for' loop
            m_resultType1 = awaiterType1.GetResult(); // Get awaiter's result

         ForLoopPrologue:
            m_x = 0;          // 'for' loop initialization
            goto ForLoopBody; // Skip to 'for' loop body

         ForLoopEpilog:
            m_resultType2 = awaiterType2.GetResult();
            m_x++;            // Increment x after each loop iteration
            // Fall into the 'for' loop's body

         ForLoopBody:
            if (m_x < 3) {  // 'for' loop test
               // Call Method2Async and get its awaiter
               awaiterType2 = Method2Async().GetAwaiter();
               if (!awaiterType2.IsCompleted) {
                  m_state = 1;                   // 'Method2Async' is completing asynchronously
                  m_awaiterType2 = awaiterType2; // Save the awaiter for when we come back

                  // Tell awaiter to call MoveNext when operation completes
                  m_builder.AwaitUnsafeOnCompleted(ref awaiterType2, ref this);
                  executeFinally = false;        // We're not logically leaving the 'try' block
                  return;                        // Thread returns to caller
               }
               // 'Method2Async' completed synchronously
               goto ForLoopEpilog;  // Completed synchronously, loop around
            }
         }
         catch (Exception) {
            Console.WriteLine("Catch");
         }
         finally {
            // Whenever a thread physically leaves a 'try', the 'finally' executes
            // We only want to execute this code when the thread logically leaves the 'try'
            if (executeFinally) {
               Console.WriteLine("Finally");
            }
         }
         result = "Done"; // What we ultimately want to return from the async function
      }
      catch (Exception exception) {
         // Unhandled exception: complete state machine's Task with exception
         m_builder.SetException(exception);
         return;
      }
      // No exception: complete state machine's Task with result
      m_builder.SetResult(result);
   }
}
```

If you spend the time to walk through the preceding code and read all the comments, I think you'll be able to fully digest what the compiler does for you. However, there is a piece of glue that attaches the object being awaited to the state machine and I think it would be helpful if I explained how this piece of glue worked. Whenever you use the `await` operator in your code, the compiler takes the specified operand and attempts to call a `GetAwaiter` method on it. This method can be either an instance method or an extension method. The object returned from calling the `GetAwaiter` method is referred to as an awaiter. An awaiter is the glue I was referring to.

After the state machine obtains an awaiter, it queries its `IsCompleted` property. If the operation completed synchronously, `true` is returned and, as an optimization, the state machine simply continues executing. At this point, it calls the awaiter's `GetResult` method, which either throws an exception if the operation failed or returns the result if the operation was successful. The state machine continues running from here to process the result.

If the operation completes asynchronously, `IsCompleted` returns `false`. In this case, the state machine calls the awaiter's `OnCompleted` method passing it a delegate to the state machine's `MoveNext` method. And now, the state machine allows its thread to return back to where it came from so that it can execute other code. In the future, the awaiter, which wraps the underlying `Task`, knows when it completes and invokes the delegate causing `MoveNext` to execute. The fields within the state machine are used to figure out how to get to the right point in the code, giving the illusion that the method is continuing from where it left off. At this point, the code calls the awaiter's `GetResult` method and execution continues running from here to process the result.

That is how async functions work and the whole purpose is to simplify the coding effort normally involved when writing non-blocking code.

Async Function Extensibility

As for extensibility, if you can wrap a `Task` object around an operation that completes in the future, you can use the `await` operator to await that operation. Having a single type (`Task`) to represent all kinds of asynchronous operations is phenomenally useful because it allows you to implement combinators (like `Task`'s `WhenAll` and `WhenAny` methods) and other helpful operations. Later in this chapter, I demonstrate doing this by wrapping a `CancellationToken` with a `Task` so I can await an asynchronous operation while also exposing timeout and cancellation.

I'd also like to share with you another example. The following is my `TaskLogger` class, which you can use to show you asynchronous operations that haven't yet completed. This is very useful in debugging scenarios especially when your application appears hung due to a bad request or a non-responding server.

```
public static class TaskLogger {
   public enum TaskLogLevel { None, Pending }
   public static TaskLogLevel LogLevel { get; set; }

   public sealed class TaskLogEntry {
      public Task Task { get; internal set; }
      public String Tag { get; internal set; }
      public DateTime LogTime { get; internal set; }
      public String CallerMemberName { get; internal set; }
      public String CallerFilePath { get; internal set; }
      public Int32 CallerLineNumber { get; internal set; }
      public override string ToString() {
         return String.Format("LogTime={0}, Tag={1}, Member={2}, File={3}({4})",
            LogTime, Tag ?? "(none)", CallerMemberName, CallerFilePath, CallerLineNumber);
      }
   }


   private static readonly ConcurrentDictionary<Task, TaskLogEntry> s_log =
      new ConcurrentDictionary<Task, TaskLogEntry>();
   public static IEnumerable<TaskLogEntry> GetLogEntries() { return s_log.Values; }

   public static Task<TResult> Log<TResult>(this Task<TResult> task, String tag = null,
      [CallerMemberName] String callerMemberName = null,
      [CallerFilePath] String callerFilePath = null,
      [CallerLineNumber] Int32 callerLineNumber = -1) {
      return (Task<TResult>)
         Log((Task)task, tag, callerMemberName, callerFilePath, callerLineNumber);
   }

   public static Task Log(this Task task, String tag = null,
      [CallerMemberName] String callerMemberName = null,
      [CallerFilePath] String callerFilePath = null,
      [CallerLineNumber] Int32 callerLineNumber = -1) {
      if (LogLevel == TaskLogLevel.None) return task;
      var logEntry = new TaskLogEntry {
         Task = task,
         LogTime = DateTime.Now,
         Tag = tag,
         CallerMemberName = callerMemberName,
         CallerFilePath = callerFilePath,
         CallerLineNumber = callerLineNumber
      };
      s_log[task] = logEntry;
      task.ContinueWith(t => { TaskLogEntry entry; s_log.TryRemove(t, out entry); },
         TaskContinuationOptions.ExecuteSynchronously);
      return task;
   }
}
```

And here is some code that demonstrates the use of the class.

```
public static async Task Go() {
#if DEBUG
   // Using TaskLogger incurs a memory and performance hit; so turn it on in debug builds
   TaskLogger.LogLevel = TaskLogger.TaskLogLevel.Pending;
#endif

   // Initiate 3 task; for testing the TaskLogger, we control their duration explicitly
   var tasks = new List<Task> {
      Task.Delay(2000).Log("2s op"),
      Task.Delay(5000).Log("5s op"),
      Task.Delay(6000).Log("6s op")
   };

   try {
      // Wait for all tasks but cancel after 3 seconds; only 1 task should complete in time
      // Note: WithCancellation is my extension method described later in this chapter
      await Task.WhenAll(tasks).
         WithCancellation(new CancellationTokenSource(3000).Token);
   }
   catch (OperationCanceledException) { }

   // Ask the logger which tasks have not yet completed and sort
   // them in order from the one that's been waiting the longest
   foreach (var op in TaskLogger.GetLogEntries().OrderBy(tle => tle.LogTime))
      Console.WriteLine(op);
}
```

When I build and run this code, I get the following output.

```
LogTime=7/16/2012 6:44:31 AM, Tag=6s op, Member=Go, File=C:\CLR via C#\Code\Ch28-1-IOOps.cs(332)
LogTime=7/16/2012 6:44:31 AM, Tag=5s op, Member=Go, File=C:\CLR via C#\Code\Ch28-1-IOOps.cs(331)
```

In addition to all the flexibility you have with using `Task`, async functions have another extensibility point: the compiler calls `GetAwaiter` on whatever operand is used with `await`. So, the operand doesn't have to be a `Task` object at all; it can be of any type as long as it has a `GetAwaiter` method available to call. Here is an example of my own awaiter that is the glue between an async method's state machine and an event being raised.

```
public sealed class EventAwaiter<TEventArgs> : INotifyCompletion {
   private ConcurrentQueue<TEventArgs> m_events = new ConcurrentQueue<TEventArgs>();
   private Action m_continuation;

   #region Members invoked by the state machine
   // The state machine will call this first to get our awaiter; we return ourself
   public EventAwaiter<TEventArgs> GetAwaiter() { return this; }

   // Tell state machine if any events have happened yet
   public Boolean IsCompleted { get { return m_events.Count > 0; } }

   // The state machine tells us what method to invoke later; we save it
   public void OnCompleted(Action continuation) {
      Volatile.Write(ref m_continuation, continuation);
   }

   // The state machine queries the result; this is the await operator's result
   public TEventArgs GetResult() {
      TEventArgs e;
      m_events.TryDequeue(out e);
      return e;
   }
   #endregion

   // Potentially invoked by multiple threads simultaneously when each raises the event
   public void EventRaised(Object sender, TEventArgs eventArgs) {
      m_events.Enqueue(eventArgs);   // Save EventArgs to return it from GetResult/await

      // If there is a pending continuation, this thread takes it
      Action continuation = Interlocked.Exchange(ref m_continuation, null);
      if (continuation != null) continuation();   // Resume the state machine
   }
}
```

And here is a method that uses my EventAwaiter class to return from an await operator whenever an event is raised. In this case, the state machine continues whenever any thread in the AppDomain throws an exception.

```
private static async void ShowExceptions() {
   var eventAwaiter = new EventAwaiter<FirstChanceExceptionEventArgs>();
   AppDomain.CurrentDomain.FirstChanceException += eventAwaiter.EventRaised;

   while (true) {
      Console.WriteLine("AppDomain exception: {0}",
         (await eventAwaiter).Exception.GetType());
   }
}
```

And finally, here is some code that demonstrates it all working.

```
public static void Go() {
   ShowExceptions();

   for (Int32 x = 0; x < 3; x++) {
      try {
         switch (x) {
            case 0: throw new InvalidOperationException();
            case 1: throw new ObjectDisposedException("");
            case 2: throw new ArgumentOutOfRangeException();
         }
      }
      catch { }
   }
}
```

Async Functions and Event Handlers

Async functions usually have a return type of either Task or Task<TResult> to represent the completion of the function's state machine. However, it is also possible to define an async function with a void return type. This is a special case that the C# compiler allows to simplify the very common scenario where you want to implement an asynchronous event handler.

Almost all event handler methods adhere to a method signature similar to this.

```
void EventHandlerCallback(Object sender, EventArgs e);
```

But, it is common to want to perform I/O operations inside an event handler, for example, when a user clicks a UI element to open a file and read from it. To keep the UI responsive, this I/O should be done asynchronously. Allowing you to write this code in an event handler method that has a `void` return type requires that the C# compiler allows async functions to have a `void` return type so you can use the `await` operator to perform non-blocking I/O operations. When an async function has a `void` return type, the compiler still generates code to create the state machine but it does not create a `Task` object because there is no way that one could be used. Because of this, there is no way to know when the state machine of a `void`-returning async function has run to completion.[5]

Async Functions in the Framework Class Library

Personally, I love async functions because they are relatively easy to learn, simple to use, and they are supported by many types in the FCL. It is easy to identify async functions because, by convention, `Async` is suffixed onto the method's name. In the Framework Class Library (FCL), many of the types that offer I/O operations offer `XxxAsync` methods.[6] Here are some examples.

- All the `System.IO.Stream`-derived classes offer `ReadAsync`, `WriteAsync`, `FlushAsync`, and `CopyToAsync` methods.
- All the `System.IO.TextReader`-derived classed offer `ReadAsync`, `ReadLineAsync`, `ReadToEndAsync`, and `ReadBlockAsync` methods. And the `System.IO.TextWriter`-derived classes offer `WriteAsync`, `WriteLineAsync`, and `FlushAsync` methods.
- The `System.Net.Http.HttpClient` class offers `GetAsync`, `GetStreamAsync`, `GetByteArrayAsync`, `PostAsync`, `PutAsync`, `DeleteAsync`, and many more.
- All `System.Net.WebRequest`-derived classes (including `FileWebRequest`, `FtpWebRequest`, and `HttpWebRequest`) offer `GetRequestStreamAsync` and `GetResponseAsync` methods.
- The `System.Data.SqlClient.SqlCommand` class offers `ExecuteDbDataReaderAsync`, `ExecuteNonQueryAsync`, `ExecuteReaderAsync`, `ExecuteScalarAsync`, and `ExecuteXmlReaderAsync` methods.
- Tools (such as SvcUtil.exe) that produce web service proxy types also generate `XxxAsync` methods.

[5] For this reason, if you try to mark your program's entry point method (`Main`) as async, the C# compiler issues the following error: `an entry point cannot be marked with the 'async' modifier`. If you put any `await` operators in your `Main` method, then your process's primary thread would return from `Main` as soon as the first `await` operator executes. But because code that calls `Main` can't get a `Task` to monitor it and wait for it to complete, the process would just terminate (because you're returning from `Main`), and the rest of the code in `Main` would never execute at all. Fortunately, the C# compiler considers this an error to prevent this from happening.

[6] WinRT methods follow the same naming convention and return an `IAsyncInfo` interface. Fortunately, the .NET Framework supplies extension methods that effectively cast an `IAsyncInfo` to a `Task`. For more information about using asynchronous WinRT APIs with async functions, see Chapter 25, "Interoperating with WinRT Components."

For anyone who has been working with earlier versions of the .NET Framework, you may be familiar with some other asynchronous programming models that it offered. There is the programming model that used BeginXxx and EndXxx methods along with an IAsyncResult interface. And there is the event-based programming model that also had XxxAsync methods (that did not return Task objects) and invoked event handler methods when an asynchronous operation completed. These two asynchronous programming models are now considered obsolete and the new model using Task objects is the preferred model.

While looking through the FCL, you might notice some classes that are lacking XxxAsync methods and instead only offer BeginXxx and EndXxx methods. This is mostly due to Microsoft not having the time to update these classes with the new methods. In the future, Microsoft should be enhancing these classes so that they fully support the new model. However, until they do, there is a helper method that you can use to adapt the old BeginXxx and EndXxx model to the new Task-based model.

Earlier I showed the code for a client application that makes a request over a named pipe. Let me show the server side of this code now.

```
private static async void StartServer() {
   while (true) {
      var pipe = new NamedPipeServerStream(c_pipeName, PipeDirection.InOut, -1,
         PipeTransmissionMode.Message, PipeOptions.Asynchronous | PipeOptions.WriteThrough);

      // Asynchronously accept a client connection
      // NOTE: NamedPipServerStream uses the old Asynchronous Programming Model (APM)
      // I convert the old APM to the new Task model via TaskFactory's FromAsync method
      await Task.Factory.FromAsync(pipe.BeginWaitForConnection, pipe.EndWaitForConnection,
         null);

      // Start servicing the client, which returns immediately because it is asynchronous
      ServiceClientRequestAsync(pipe);
   }
}
```

The NamedPipeServerStream class has BeginWaitForConnection and EndWaitForConnection methods defined, but it does not yet have a WaitForConnectionAsync method defined. Hopefully this method will be added in a future version of the FCL. However, all is not lost, because, as you see in the preceding code, I call TaskScheduler's FromAsync method, passing into it the names of the BeginXxx and EndXxx methods, and then FromAsync internally creates a Task object that wraps these methods. Now I can use the Task object with the await operator.[7]

[7] TaskScheduler's FromAsync method has overloads that accept an IAsyncResult in addition to overloads that accept delegates to the BeginXxx and EndXxx methods. When possible, avoid the overloads that accept an IAsyncResult because they are less efficient.

For the old event-based programming model, the FCL does not include any helper methods to adapt this model into the new `Task`-based model. So you have to hand-code it. Here is code demonstrating how to wrap a `WebClient` (which uses the event-based programming model) with a `TaskCompletionSource` so it can be `await`ed on in an async function.

```
private static async Task<String> AwaitWebClient(Uri uri) {
   // The System.Net.WebClient class supports the Event-based Asynchronous Pattern
   var wc = new System.Net.WebClient();

   // Create the TaskCompletionSource and its underlying Task object
   var tcs = new TaskCompletionSource<String>();

   // When a string completes downloading, the WebClient object raises the
   // DownloadStringCompleted event, which completes the TaskCompletionSource
   wc.DownloadStringCompleted += (s, e) => {
      if (e.Cancelled) tcs.SetCanceled();
      else if (e.Error != null) tcs.SetException(e.Error);
      else tcs.SetResult(e.Result);
   };

   // Start the asynchronous operation
   wc.DownloadStringAsync(uri);

   // Now, we can the TaskCompletionSource's Task and process the result as usual
   String result = await tcs.Task;
   // Process the resulting string (if desired)...

   return result;
}
```

Async Functions and Exception Handling

When a Windows device driver is processing an asynchronous I/O request, it is possible for something to go wrong, and Windows will need to inform your application of this. For example, while sending bytes or waiting for bytes to come in over the network, a timeout could expire. If the data does not come in time, the device driver will want to tell you that the asynchronous operation completed with an error. To accomplish this, the device driver posts the completed IRP to the CLR's thread pool and a thread pool thread will complete the `Task` object with an exception. When your state machine method is resumed, the `await` operator sees that the operation failed and throws this exception.

In Chapter 27, I discussed how `Task` objects normally throw an `AggregateException`, and then you'd query this exception's `InnerExceptions` property to see the real exception(s) that occurred. However, when using `await` with a `Task`, the first inner exception is thrown instead of an `AggregateException`.[8] This was done to give you the programming experience you expect. Also, without this, you'd have to catch `AggregateException` throughout your code, check the inner exception and either handle the exception or re-throw it. This would be very cumbersome.

[8] For the curious, it is `TaskAwaiter`'s `GetResult` method that throws the first inner exception instead of throwing an `AggregateException`.

If your state machine method experiences an unhandled exception, then the Task object representing your async function completes due to the unhandled exception. Any code waiting for this Task object to complete will see the exception. However, it is also possible for an async function to have a void return type. In this case, there is no way for a caller to discover the unhandled exception. So, when a void-returning async function throws an unhandled exception, the compiler-generated code catches it and causes it to be rethrown using the caller's synchronization context (discussed later). If the caller executed via a GUI thread, the GUI thread will eventually rethrow the exception. If the caller executed via a non-GUI thread, some thread pool thread will eventually rethrow the exception. Usually, rethrowing these exceptions causes the whole process to terminate.

Other Async Function Features

In this section, I'd like to share with you some additional features related to async functions. Microsoft Visual Studio has great support for debugging async functions. When the debugger is stopped on an await operator, stepping over (F10) will actually break into the debugger when the next statement is reached after the operation completes. This code might even execute on a different thread than the one that initiated the operation! This is incredibly useful and simplifies debugging substantially.

Also, if you accidentally step into (F11), an async function, you can step out (Shift+F11) of the function to get back to the caller; you must do this while on the opening brace of the async function. After you pass the open brace, step out (Shift+F11) won't break until the async function runs all the way to completion. If you need to debug the calling method before the state machine runs to completion, put a breakpoint in the calling method and just run (F5) to it.

Some asynchronous operations execute quickly and therefore complete almost instantaneously. When this happens, it is inefficient to suspend the state machine and then have another thread immediately resume the state machine; it is much more efficient to have the state machine just continue its execution. Fortunately, the await operator's compiler-generated code does check for this. If an asynchronous operation completes just before the thread would return, the thread does not return and instead, it just executes the next line of code.

This is all fine and good but occasionally, you might have an async function that performs an intensive amount of processing before initiating an asynchronous operation. If you invoke the function via your app's GUI thread, your user interface will become non-responsive to the user. And, if the asynchronous operation completes synchronously, then your user-interface will be non-responsive even longer. So, if you want to initiate an async function from a thread other than the thread that calls it, you can use Task's static Run method as follows.

```
// Task.Run is called on the GUI thread
Task.Run(async () => {
   // This code runs on a thread pool thread
   // TODO: Do intensive compute-bound processing here...

   await XxxAsync();  // Initiate asynchronous operation
   // Do more processing here...
});
```

This code demonstrates another C# feature: async lambda expressions. You see, you can't just put an await operator inside the body of a regular lambda expression, because the compiler wouldn't know how to turn the method into a state machine. But placing async just before the lambda expression causes the compiler to turn the lambda expression into a state machine method that returns a Task or Task<TResult>, which can then be assigned to any Func delegate variable whose return type is Task or Task<TResult>.

When writing code, it is very easy to invoke an async function, forgetting to use the await operator; the following code demonstrates.

```
static async Task OuterAsyncFunction() {
   InnerAsyncFunction();     // Oops, forgot to put the await operator on this line!

   // Code here continues to execute while InnerAsyncFunction also continues to execute...
}

static async Task InnerAsyncFunction() { /* Code in here not important */ }
```

Fortunately, when you do this, the C# compiler issues the following warning: `Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'await' operator to the result of the call.` This is nice but on rare occasions, you actually don't care when InnerAsyncFunction completes, and you do want to write the preceding code and not have the compiler issue a warning.

To quiet the compiler warning, you can simply assign the Task returned from InnerAsyncFunction to a variable and then ignore the variable.[9]

```
static async Task OuterAsyncFunction() {
   var noWarning = InnerAsyncFunction(); // I intend not to put the await operator on this line.

   // Code here continues to execute while InnerAsyncFunction also continues to execute...
}
```

Or, I prefer to define an extension method that looks like this.

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]  // Causes compiler to optimize the call away
public static void NoWarning(this Task task) { /* No code goes in here */ }
```

And then I can use it like this.

```
static async Task OuterAsyncFunction() {
   InnerAsyncFunction().NoWarning(); // I intend not to put the await operator on this line.

   // Code here continues to execute while InnerAsyncFunction also continues to execute...
}
```

One of the truly great features of performing asynchronous I/O operations is that you can initiate many of them concurrently so that they are all executing in parallel. This can give your application

[9] Fortunately, the compiler does not give a warning about the local variable never being used.

a phenomenal performance boost. I never showed you the code that started my named pipe server and then made a bunch of client requests to it. Let me show that code now.

```
public static async Task Go() {
   // Start the server, which returns immediately because
   // it asynchronously waits for client requests
   StartServer(); // This returns void, so compiler warning to deal with

   // Make lots of async client requests; save each client's Task<String>
   List<Task<String>> requests = new List<Task<String>>(10000);
   for (Int32 n = 0; n < requests.Capacity; n++)
      requests.Add(IssueClientRequestAsync("localhost", "Request #" + n));

   // Asynchronously wait until all client requests have completed
   // NOTE: If 1+ tasks throws, WhenAll rethrows the last-throw exception
   String[] responses = await Task.WhenAll(requests);

   // Process all the responses
   for (Int32 n = 0; n < responses.Length; n++)
      Console.WriteLine(responses[n]);
}
```

This code starts the named pipe server so that it is listening for client requests and then, in a `for` loop, it initiates 10,000 client requests as fast as it possibly can. Each time `IssueClientRequest-Async` is called, it returns a `Task<String>` object, which I then add to a collection. Now, the named pipe server is processing these client requests as fast as it possibly can using thread pool threads that will try to keep all the CPUs on the machine busy.[10] As the server completes processing each request; each request's `Task<String>` object completes with the string response returned from the server.

In the preceding code, I want to wait until all the client requests have gotten their response before processing their results. I accomplish this by calling `Task`'s static `WhenAll` method. Internally, this method creates a `Task<String[]>` object that completes after all of the `List`'s `Task` objects have completed. I then `await` the `Task<String[]>` object so that the state machine continues execution after all of the tasks have completed. After all the tasks have completed, I loop through all the responses at once and process them (call `Console.WriteLine`).

Perhaps you'd prefer to process each response as it happens rather than waiting for all of them to complete. Accomplishing this is almost as easy by way of `Task`'s static `WhenAny` method. The revised code looks like this.

```
public static async Task Go() {
   // Start the server, which returns immediately because
   // it asynchronously waits for client requests
   StartServer();

   // Make lots of async client requests; save each client's Task<String>
   List<Task<String>> requests = new List<Task<String>>(10000);
```

10 Fun observation: when I tested this code on my machine, the CPU usage on my 8-processor machine went all the way up to 100 percent, of course. Because all the CPUs were busy, the machine got hotter and the fan got a lot louder! After processing completed, the CPU usage went down and the fan got quieter. Fan volume is a new way of verifying that everything is working as it should.

```
   for (Int32 n = 0; n < requests.Capacity; n++)
      requests.Add(IssueClientRequestAsync("localhost", "Request #" + n));

   // Continue AS EACH task completes
   while (requests.Count > 0) {
      // Process each completed response sequentially
      Task<String> response = await Task.WhenAny(requests);
      requests.Remove(response);  // Remove the completed task from the collection

      // Process a single client's response
      Console.WriteLine(response.Result);
   }
}
```

Here, I create a `while` loop that iterates once per client request. Inside the loop I `await Task`'s `WhenAny` method, which returns one `Task<String>` object at a time, indicating a client request that has been responded to by the server. After I get this `Task<String>` object, I remove it from the collection, and then I query its result in order to process it (pass it to `Console.WriteLine`).

Applications and Their Threading Models

The .NET Framework supports several different kinds of application models, and each application model can impose its own threading model. Console applications and Windows Services (which are really console applications; you just don't see the console) do not impose any kind of threading model; that is, any thread can do whatever it wants when it wants.

However, GUI applications, including Windows Forms, Windows Presentation Foundation (WPF), Silverlight, and Windows Store apps impose a threading model where the thread that created a UI element is the only thread allowed to update that UI element. It is common for the GUI thread to spawn off an asynchronous operation so that the GUI thread doesn't block and stop responding to user input like mouse, keystroke, pen, and touch events. However, when the asynchronous operation completes, a thread pool thread completes the `Task` object resuming the state machine.

For some application models, this is OK and even desired because it's efficient. But for some other application models, like GUI applications, this is a problem, because your code will throw an exception if it tries to update UI elements via a thread pool thread. Somehow, the thread pool thread must have the GUI thread update the UI elements.

ASP.NET applications allow any thread to do whatever it wants. When a thread pool thread starts to process a client's request, it can assume the client's culture (`System.Globalization.CultureInfo`), allowing the web server to return culture-specific formatting for numbers, dates, and times.[11] In addition, the web server can assume the client's identity (`System.Security.Principal.IPrincipal`), so that the server can access only the resources that the client is allowed to access. When a thread pool thread spawns an asynchronous operation, it may be completed by another thread pool thread, which will be processing the result of an asynchronous operation. While this

[11] For more information, see *http://msdn.microsoft.com/en-us/library/bz9tc508.aspx*.

work is being performed on behalf of the original client request, the culture and identity needs to "flow" to the new thread pool thread so any additional work done on behalf of the client is performed using the client's culture and identity information.

Fortunately, the FCL defines a base class, called `System.Threading.SynchronizationContext`, which solves all these problems. Simply stated, a `SynchronizationContext`-derived object connects an application model to its threading model. The FCL defines several classes derived from `SynchronizationContext`, but usually you will not deal directly with these classes; in fact, many of them are not publicly exposed or documented.

For the most part, application developers do not need to know anything about the `SynchronizationContext` class. When you `await` a `Task`, the calling thread's `SynchronizationContext` object is obtained. When a thread pool thread completes the `Task`, the `SynchronizationContext` object is used, ensuring the right threading model for your application model. So, when a GUI thread awaits a `Task`, the code following the `await` operator is guaranteed to execute on the GUI thread as well, allowing that code to update UI elements.[12] For an ASP.NET application, the code following the `await` operator is guaranteed to execute on a thread pool thread that has the client's culture and principal information associated with it.

Most of the time, having a state machine resume using the application model's threading model is phenomenally useful and convenient. But, on some occasions, this can get you into trouble. Here is an example that causes a WPF application to deadlock.

```
private sealed class MyWpfWindow : Window {
   public MyWpfWindow() { Title = "WPF Window"; }

   protected override void OnActivated(EventArgs e) {
      // Querying the Result property prevents the GUI thread from returning;
      // the thread blocks waiting for the result
      String http = GetHttp().Result;  // Get the string synchronously!

      base.OnActivated(e);
   }

   private async Task<String> GetHttp() {
      // Issue the HTTP request and let the thread return from GetHttp
      HttpResponseMessage msg = await new HttpClient().GetAsync("http://Wintellect.com/");
      // We never get here: The GUI thread is waiting for this method to finish but this method
      // can't finish because the GUI thread is waiting for it to finish --> DEADLOCK!

      return await msg.Content.ReadAsStringAsync();
   }
}
```

Developers creating class libraries definitely need to be aware of the `SynchronizationContext` class so they can write high-performance code that works with all application models. Because a lot of class library code is application model agnostic, we want to avoid the additional overhead involved

12 Internally, the various `SynchronizationContext`-derived classes get the GUI thread to resume the state machine using methods like `System.Windows.Forms.Control.BeginInvoke`, `System.Windows.Threading.Dispatcher.BeginInvoke`, and `Windows.UI.Core.CoreDispatcher.RunAsync`.

in using a SynchronizationContext object. In addition, class library developers should do everything in their power to help application developers avoid deadlock situations. To solve both of these problems, both the Task and Task<TResult> classes offer a method called ConfigureAwait whose signature looks like this.

```
// Task defines this method:
public ConfiguredTaskAwaitable          ConfigureAwait(Boolean continueOnCapturedContext);

// Task<TResult> defines this method:
public ConfiguredTaskAwaitable<TResult> ConfigureAwait(Boolean continueOnCapturedContext);
```

Passing true to this method gives you the same behavior as not calling the method at all. But, if you pass false, the await operator does not query the calling thread's SynchronizationContext object and, when a thread pool thread completes the Task, it simply completes it and the code after the await operator executes via the thread pool thread.

Even though my GetHttp method is not class library code, the deadlock problem goes away if I add calls to ConfigureAwait. Here is the modified version of my GetHttp method.

```
private async Task<String> GetHttp() {
   // Issue the HTTP request and let the thread return from GetHttp
   HttpResponseMessage msg = await new HttpClient().GetAsync("http://Wintellect.com/")
      .ConfigureAwait(false);
   // We DO get here now because a thread pool can execute this code
   // as opposed to forcing the GUI thread to execute it.

   return await msg.Content.ReadAsStringAsync().ConfigureAwait(false);
}
```

As the preceding code shows, ConfigureAwait(false) must be applied to every Task object you await. This is because asynchronous operations may complete synchronously and, when this happens, the calling thread simply continues executing without returning to its caller; you never know which operation requires ignoring the SynchronizationContext object, so you have to tell all of them to ignore it. This also means that your class library code should be application model agnostic.

Alternatively, I could re-write my GetHttp method as follows so that the whole thing executes via a thread pool thread.

```
private Task<String> GetHttp() {
   return Task.Run(async () => {
      // We run on a thread pool thread now that has no SynchronizationContext on it
      HttpResponseMessage msg = await new HttpClient().GetAsync("http://Wintellect.com/");
      // We DO get here because some thread pool can execute this code

      return await msg.Content.ReadAsStringAsync();
   });
}
```

In this version of the code, notice that my GetHttp method is not an async function; I removed the async keyword from the method signature, because the method no longer has an await operator in it. On the other hand, the lambda expression I pass to Task.Run is an async function.

Implementing a Server Asynchronously

From talking to many developers over the years, I've discovered that very few of them are aware that the .NET Framework has built-in support allowing you to build asynchronous servers that scale really well. In this book, I can't explain how to do this for every kind of server, but I can list what you should look for in the MSDN documentation.

- To build asynchronous ASP.NET Web Forms: in your .aspx file, add "Async=true" to your page directive and look up the `System.Web.UI.Page`'s `RegisterAsyncTask` method.
- To build an asynchronous ASP.NET MVC controller: derive your controller class from `System.Web.Mvc.AsyncController` and simply have your action method return a `Task<ActionResult>`.
- To build an asynchronous ASP.NET handler: derive your class from `System.Web.HttpTaskAsyncHandler` and then override its abstract `ProcessRequestAsync` method.
- To build an asynchronous WCF service: implement your service as an async function and have it return `Task` or `Task<TResult>`.

Canceling I/O Operations

In general, Windows doesn't give you a way to cancel an outstanding I/O operation. This is a feature that many developers would like, but it is actually quite hard to implement. After all, if you make a request from a server and then you decide you don't want the response anymore, there is no way to tell the server to abandon your original request. The way to deal with this is just to let the bytes come back to the client machine and then throw them away. In addition, there is a race condition here—your request to cancel the request could come just as the server is sending the response. Now what should your application do? You'd need to handle this potential race condition occurring in your own code and decide whether to throw the data away or act on it.

To assist with this, I recommend you implement a `WithCancellation` extension method that extends `Task<TResult>` (you need a similar overload that extends `Task` too) as follows.

```
private struct Void { } // Because there isn't a non-generic TaskCompletionSource class.

private static async Task<TResult> WithCancellation<TResult>(this Task<TResult> originalTask,
   CancellationToken ct) {

   // Create a Task that completes when the CancellationToken is canceled
   var cancelTask = new TaskCompletionSource<Void>();

   // When the CancellationToken is canceled, complete the Task
   using (ct.Register(
      t => ((TaskCompletionSource<Void>)t).TrySetResult(new Void()), cancelTask)) {

      // Create a Task that completes when either the original or
      // CancellationToken Task completes
      Task any = await Task.WhenAny(originalTask, cancelTask.Task);

      // If any Task completes due to CancellationToken, throw OperationCanceledException
      if (any == cancelTask.Task) ct.ThrowIfCancellationRequested();
   }

   // await original task (synchronously); if it failed, awaiting it
   // throws 1st inner exception instead of AggregateException
   return await originalTask;
}
```

Now, you can call this extension method as follows.

```
public static async Task Go() {
   // Create a CancellationTokenSource that cancels itself after # milliseconds
   var cts = new CancellationTokenSource(5000); // To cancel sooner, call cts.Cancel()
   var ct = cts.Token;

   try {
      // I used Task.Delay for testing; replace this with another method that returns a Task
      await Task.Delay(10000).WithCancellation(ct);
      Console.WriteLine("Task completed");
   }
   catch (OperationCanceledException) {
      Console.WriteLine("Task cancelled");
   }
}
```

Some I/O Operations Must Be Done Synchronously

The Win32 API offers many functions that execute I/O operations. Unfortunately, some of these methods do not let you perform the I/O asynchronously. For example, the Win32 `CreateFile` method (called by `FileStream`'s constructor) always executes synchronously. If you're trying to create or open a file on a network server, it could take several seconds before `CreateFile` returns—the calling thread is idle all the while. An application designed for optimum responsiveness and scalability would ideally call a Win32 function that lets you create or open a file asynchronously so that your thread is not sitting and waiting for the server to reply. Unfortunately, Win32 has no `CreateFile`-like function to let you do this, and therefore the FCL cannot offer an efficient way to open a file asynchronously. Windows also doesn't offer functions to asynchronously access the registry, access the event log, get a directory's files/subdirectories, or change a file's/directory's attributes, to name just a few.

Here is an example where this is a real problem. Imagine writing a simple UI control that allows the user to type a file path and provides automatic completion (similar to the common File Open dialog box). The control must use separate threads to enumerate directories looking for files because Windows doesn't offer any functions to enumerate files asynchronously. As the user continues to type in the UI control, you have to use more threads and ignore the results from any previously spawned threads. With Windows Vista, Microsoft introduced a new Win32 function called `CancelSynchronousIO`. This function allows one thread to cancel a synchronous I/O operation that is being performed by another thread. This function is not exposed by the FCL, but you can also P/Invoke to it if you want to take advantage of it from a desktop application implemented with managed code. I show the P/Invoke signature near the end of this chapter.

The point I want you to take away though is that many people think that synchronous APIs are easier to work with, and in many cases this is true. But in some cases, synchronous APIs make things much harder.

Due to all the problems that exist when executing I/O operations synchronously, when designing the Windows Runtime, the Windows team decided to expose all methods that perform I/O asynchronously. So, now there is a Windows Runtime API to open files asynchronously; see `Windows.Storage.StorageFile`'s `OpenAsync` method. In fact, the Windows Runtime does not offer any APIs allowing you to perform an I/O operation synchronously. Fortunately, you can use the C#'s async function feature to simplify your coding when calling these APIs.

`FileStream`-Specific Issues

When you create a `FileStream` object, you get to specify whether you want to communicate using synchronous or asynchronous operations via the `FileOptions.Asynchronous` flag (which is equivalent to calling the Win32 `CreateFile` function and passing into it the `FILE_FLAG_OVERLAPPED` flag). If you do not specify this flag, Windows performs all operations against the file synchronously. Of course, you can still call `FileStream`'s `ReadAsync` method, and to your application, it looks as if the operation is being performed asynchronously, but internally, the `FileStream` class uses another thread to emulate asynchronous behavior; use of this thread is wasteful and hurts performance.

On the other hand, you can create a `FileStream` object by specifying the `FileOptions.Asynchronous` flag. Then you can call `FileStream`'s `Read` method to perform a synchronous operation. Internally, the `FileStream` class emulates this behavior by starting an asynchronous operation and then immediately puts the calling thread to sleep until the operation is complete. This is also inefficient, but it is not as inefficient as calling `ReadAsync` by using a `FileStream` constructed without the `FileOptions.Asynchronous` flag.

So, to summarize, when working with a `FileStream`, you must decide up front whether you intend to perform synchronous or asynchronous I/O against the file and indicate your choice by specifying the `FileOptions.Asynchronous` flag (or not). If you specify this flag, always call `ReadAsync`. If you do not specify this flag, always call `Read`. This will give you the best performance. If you intend to make some synchronous and some asynchronous operations against the `FileStream`, it is more efficient to construct it using the `FileOptions.Asynchronous` flag. Alternatively, you can create two `FileStream` objects over the same file; open one `FileStream` for asynchronous I/O and open the other `FileStream` for synchronous I/O. Note that the `System.IO.File`'s class offers helper methods (`Create`, `Open`, and `OpenWrite`) that create and return `FileStream` objects. Internally, none of these methods specify the `FileOptions.Asynchronous` flag, so you should avoid using these methods if you want to create a responsive or scalable application.

You should also be aware that the NTFS file system device driver performs some operations synchronously no matter how you open the file. For more information about this, see *http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B156932*.

I/O Request Priorities

In Chapter 26, "Thread Basics," I showed how setting thread priorities affects how threads are scheduled. However, threads also perform I/O requests to read and write data from various hardware devices. If a low-priority thread gets CPU time, it could easily queue hundreds or thousands of I/O requests in a very short time. Because I/O requests typically require time to process, it is possible that a low-priority thread could significantly affect the responsiveness of the system by suspending high-priority threads, which prevents them from getting their work done. Because of this, you can see a machine become less responsive when executing long-running low-priority services such as disk defragmenters, virus scanners, content indexers, and so on.[13]

Windows allows a thread to specify a priority when making I/O requests. For more details about I/O priorities, refer to the white paper at *http://www.microsoft.com/whdc/driver/priorityio.mspx*. Unfortunately, the FCL does not include this functionality yet; hopefully, it will be added in a future version. However, you can still take advantage of this feature by P/Invoking out to native Win32 functions. Here is the P/Invoke code.

```
internal static class ThreadIO {
   public static BackgroundProcessingDisposer BeginBackgroundProcessing(
      Boolean process = false) {

      ChangeBackgroundProcessing(process, true);
      return new BackgroundProcessingDisposer(process);
   }

   public static void EndBackgroundProcessing(Boolean process = false) {
      ChangeBackgroundProcessing(process, false);
   }

   private static void ChangeBackgroundProcessing(Boolean process, Boolean start) {
      Boolean ok = process
         ? SetPriorityClass(GetCurrentWin32ProcessHandle(),
               start ? ProcessBackgroundMode.Start : ProcessBackgroundMode.End)
         : SetThreadPriority(GetCurrentWin32ThreadHandle(),
               start ? ThreadBackgroundgMode.Start : ThreadBackgroundgMode.End);
      if (!ok) throw new Win32Exception();
   }

   // This struct lets C#'s using statement end the background processing mode
   public struct BackgroundProcessingDisposer : IDisposable {
      private readonly Boolean m_process;
      public BackgroundProcessingDisposer(Boolean process) { m_process = process; }
      public void Dispose() { EndBackgroundProcessing(m_process); }
   }
```

13 The Windows SuperFetch feature takes advantage of low-priority I/O requests.

```
   // See Win32's THREAD_MODE_BACKGROUND_BEGIN and THREAD_MODE_BACKGROUND_END
   private enum ThreadBackgroundgMode { Start = 0x10000, End = 0x20000 }

   // See Win32's PROCESS_MODE_BACKGROUND_BEGIN and PROCESS_MODE_BACKGROUND_END
   private enum ProcessBackgroundMode { Start = 0x100000, End = 0x200000 }

   [DllImport("Kernel32", EntryPoint = "GetCurrentProcess", ExactSpelling = true)]
   private static extern SafeWaitHandle GetCurrentWin32ProcessHandle();

   [DllImport("Kernel32", ExactSpelling = true, SetLastError = true)]
   [return: MarshalAs(UnmanagedType.Bool)]
   private static extern Boolean SetPriorityClass(
      SafeWaitHandle hprocess, ProcessBackgroundMode mode);

   [DllImport("Kernel32", EntryPoint = "GetCurrentThread", ExactSpelling = true)]
   private static extern SafeWaitHandle GetCurrentWin32ThreadHandle();

   [DllImport("Kernel32", ExactSpelling = true, SetLastError = true)]
   [return: MarshalAs(UnmanagedType.Bool)]
   private static extern Boolean SetThreadPriority(
      SafeWaitHandle hthread, ThreadBackgroundgMode mode);

   // http://msdn.microsoft.com/en-us/library/aa480216.aspx
   [DllImport("Kernel32", SetLastError = true, EntryPoint = "CancelSynchronousIo")]
   [return: MarshalAs(UnmanagedType.Bool)]
   private static extern Boolean CancelSynchronousIO(SafeWaitHandle hThread);
}
```

And here is code showing how to use it.

```
public static void Main () {
   using (ThreadIO.BeginBackgroundProcessing()) {
      // Issue low-priority I/O requests in here (eg: calls to ReadAsync/WriteAsync)
   }
}
```

You tell Windows that you want your thread to issue low-priority I/O requests by calling `ThreadIO`'s `BeginBackgroundProcessing` method. Note that this also lowers the CPU scheduling priority of the thread. You can return the thread to making normal-priority I/O requests (and normal CPU scheduling priority) by calling `EndBackgroundProcessing` or by calling `Dispose` on the value returned by `BeginBackgroundProcessing` (as shown in the preceding code via C#'s `using` statement). A thread can only affect its own background processing mode; Windows doesn't allow a thread to change the background processing mode of another thread.

If you want all threads in a process to make low-priority I/O requests and have low CPU scheduling, you can call `BeginBackgroundProcessing`, passing in `true` for the `process` parameter. A process can only affect its own background processing mode; Windows doesn't allow a thread to change the background processing mode of another process.

Important As a developer, it is your responsibility to use these new background priorities to allow the foreground applications to be more responsive, taking care to avoid *priority inversion*. In the presence of intense normal-priority I/Os, a thread running at background priority can be delayed for *seconds* before getting the result of its I/O requests. If a low-priority thread has grabbed a thread synchronization lock for which the normal-priority thread is waiting, the normal-priority threads might end up waiting for the background-priority thread until the low-priority I/O requests are completed. Your background-priority thread does not even have to submit I/Os for the problem to happen. So using shared synchronization objects between normal-priority and background-priority threads should be minimized (or eliminated if possible) to avoid these priority inversions where normal-priority threads are blocked on locks owned by background-priority threads.